What every programmer should know about high performance computing

(instruction-level parallelism, memory)

Codes: `https://github.com/dmalhotra/fwam2022`

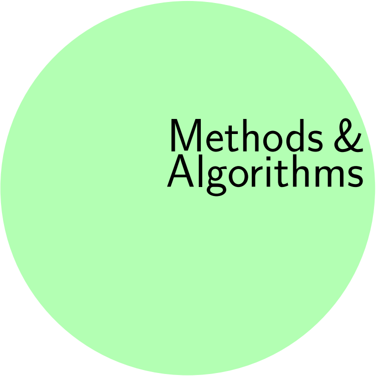
Dhairya Malhotra

$$F_\omega(\alpha + m)!$$

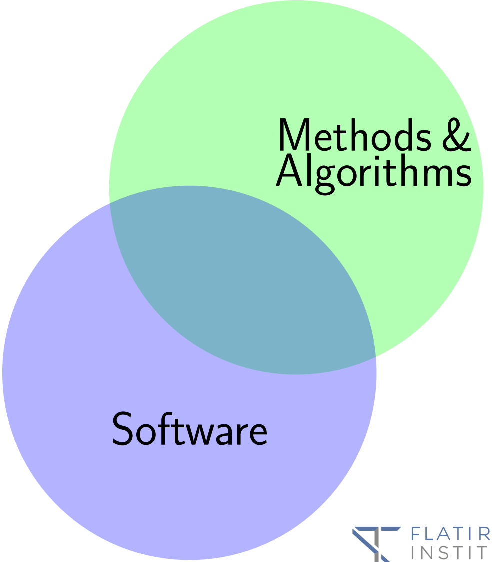
Oct 28, 2022

FLATIRON INSTITUTE

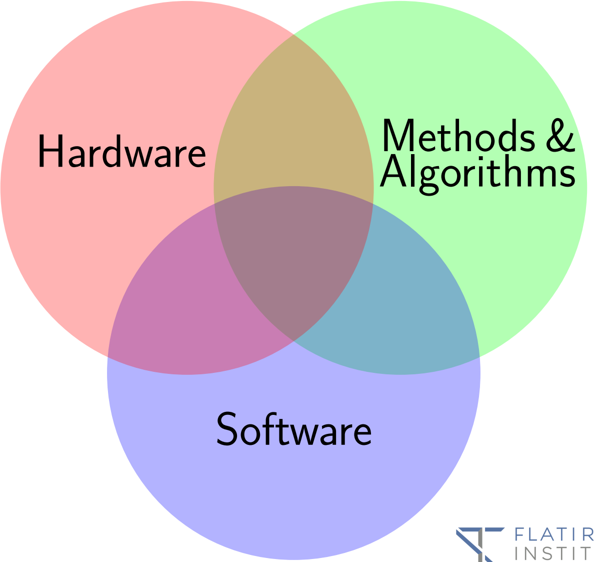
What is HPC?



What is HPC?

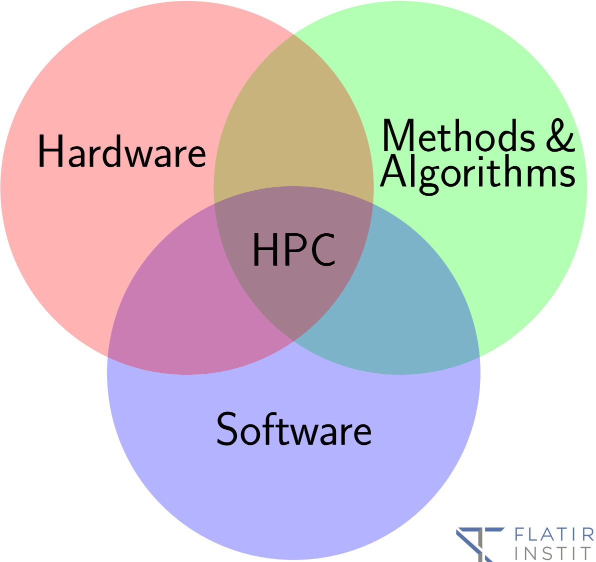


What is HPC?



What is HPC?

How can we keep our methods/algorithms and codes relevant in the future?



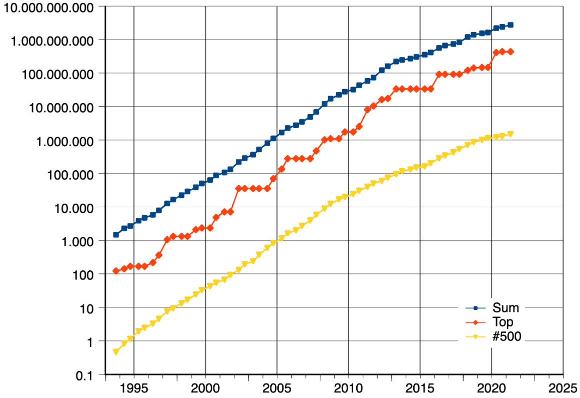
Exascale computing

- Planned
  - 2 exaFLOP Aurora supercomputer Intel Xeon Sapphire Rapids, Intel Xe GPU's
- x86 processors dominate (Intel, AMD)
  - more ARM processors recently
- GPU accelerators (7 of top 10)
  - AMD's Heterogeneous Interface for Portability (HIP)
  - NVIDIA's CUDA

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,730,112	1,102.00	1,685.65	21,100
2	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899
3	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	1,110,144	151.90	214.35	2,942
4	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148.60	200.79	10,096
5	**Sierra** - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94.64	125.71	7,438

Exascale computing

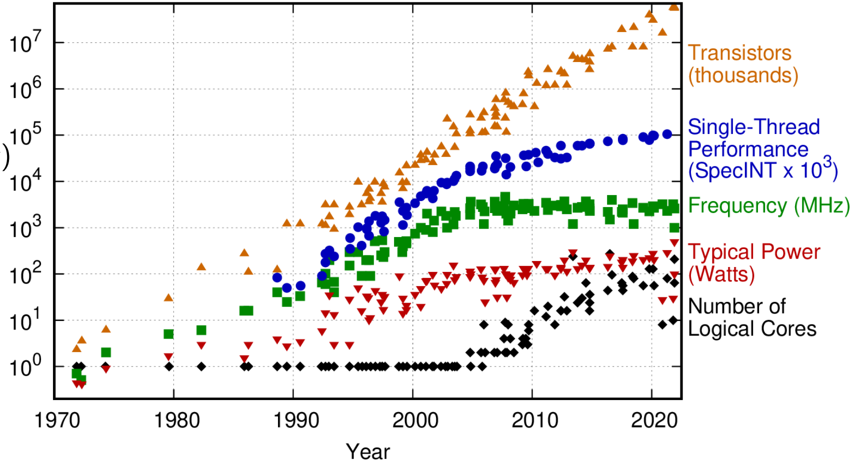
- Planned
  - 2 exaFLOP Aurora supercomputer Intel Xeon Sapphire Rapids, Intel Xe GPU's
- x86 processors dominate (Intel, AMD)
  - more ARM processors recently
- GPU accelerators (7 of top 10)
  - AMD's Heterogeneous Interface for Portability (HIP)
  - NVIDIA's CUDA



FLATIRON INSTITUTE

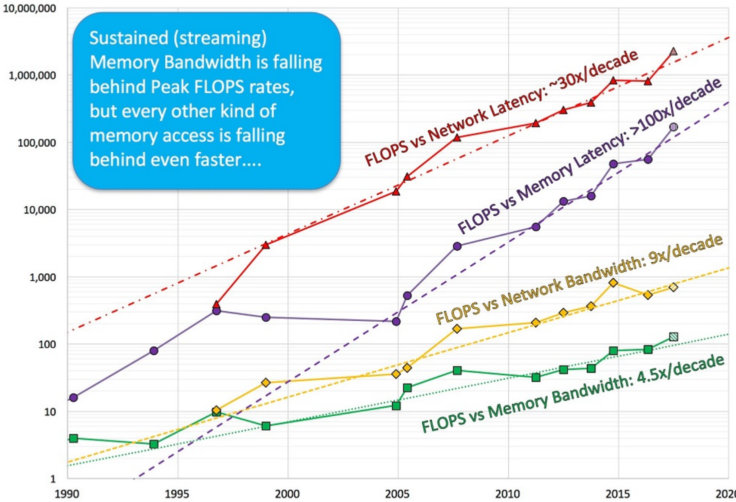
Trends in hardware

- Dennard scaling ended 2006
- Moore's law still going strong (for now)
- Multi- & many-core
- Single core performance
  - 512-bit vectors
  - superscalar,
  - pipelining
  - out-of-order ex.
  - speculative ex.



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2021 by K. Rupp

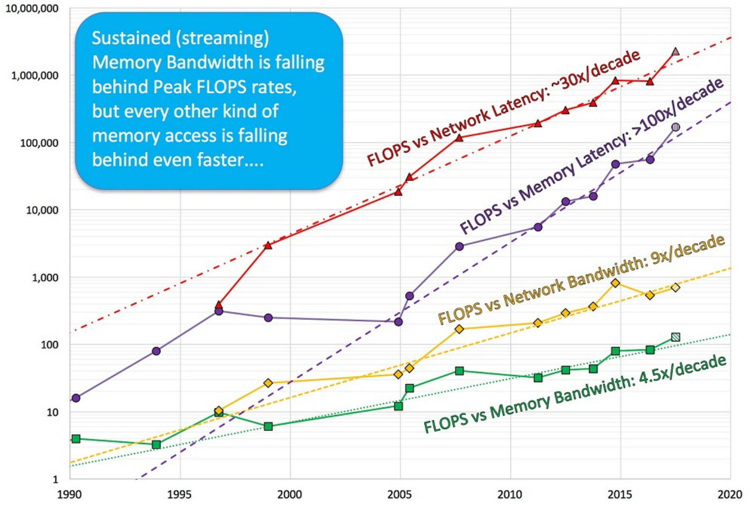
Memory wall



The situation is dire!

Source: John McCalpin - Memory bandwidth and system balance in HPC systems, 2016

Memory wall



The situation is dire!

Solutions:

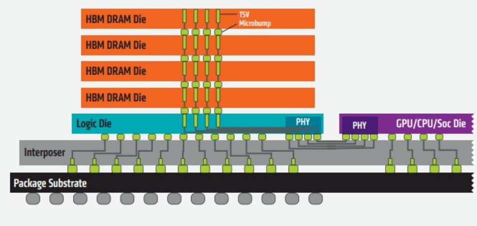
- Caches
- Non-uniform memory access (NUMA)
- High bandwidth memory (HBM)

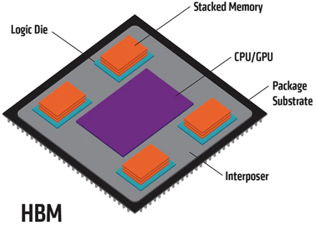
Source: John McCalpin - Memory bandwidth and system balance in HPC systems, 2016

FLATIRON INSTITUTE

High bandwidth memory

- Larger off-chip cache
- Faster on-package RAM
- Already used in many GPUs (NVIDIA, AMD)
- Fujitsu A64FX (Fugaku supercomputer)
  - HBM2: 32 GB, 1 TB/s
- Planned:
  - Intel Xeon Sapphire Rapids CPU, 2 exaFLOP Aurora supercomputer





Source: `https://www.amd.com/en/technologies/hbm`

FLATIRON INSTITUTE

Programming languages

Types of programming languages:

- Compiled: FORTRAN, C/C++, Rust
- Interpreted: Python, Julia, MATLAB

Programming languages

Types of programming languages:

- Compiled: FORTRAN, C/C++, Rust
- Interpreted: Python, Julia, MATLAB
- JIT: Julia, Numba (py), MATLAB (2015)

Programming languages

Types of programming languages:

- Compiled: FORTRAN, C/C++, Rust
- Interpreted: Python, Julia, MATLAB
- JIT: Julia, Numba (py), MATLAB (2015)

Programming languages provide an abstract view of the computer hardware.
It determines how your code executes on the hardware and how much control you have.

Programming languages

Types of programming languages:

- Compiled: FORTRAN, C/C++, Rust
- Interpreted: Python, Julia, MATLAB
- JIT: Julia, Numba (py), MATLAB (2015)

Programming languages provide an abstract view of the computer hardware.
It determines how your code executes on the hardware and how much control you have.

- Know the strengths, weaknesses and best practices for your language
  e.g. don't iterate over billion element array in python.
- Use compilation flags for best performance (e.g. for C/C++: -O3 -march=native)

Programming languages

Types of programming languages:

- Compiled: FORTRAN, C/C++, Rust
- Interpreted: Python, Julia, MATLAB
- JIT: Julia, Numba (py), MATLAB (2015)

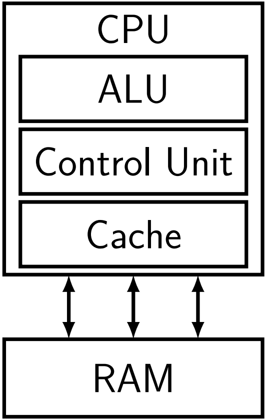
Programming languages provide an abstract view of the computer hardware.
It determines how your code executes on the hardware and how much control you have.

- Know the strengths, weaknesses and best practices for your language
  e.g. don't iterate over billion element array in python.
- Use compilation flags for best performance (e.g. for C/C++: -O3 -march=native)
- Use optimized high-performance libraries:
  - Python: NumPy, SciPy
  - MATLAB: Chebfun
  - FORTRAN, C/C++: BLAS, LAPACK, FFTW
  - many others (depending on language and field)

FLATIRON INSTITUTE

How code executes on a computer

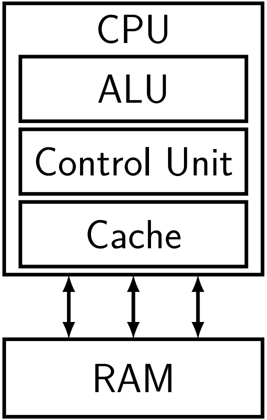
```c
void laplace(double* u, double* x,
             double* y, double* f,
                  long Ns, long Nt) {
  for (long t = 0; t < Nt; t++) {
    for (long s = 0; s < Ns; s++) {
      double rx, ry, rz;
      rx = x[s*3]-y[t*3];
      ry = x[s*3+1]-y[t*3+1];
      rz = x[s*3+2]-y[t*3+2];

      double r2 = rx*rx+ry*ry+rz*rz;
      if (r2 > 0) {
        double rinv = 1/sqrt(r2);
        u[t] += f[s] * rinv;
      }
    }
  }
}
```



- code executes line-by-line
- sequentially and in order
- one scalar operation at a time
- one operation per clock cycle

How code executes on a computer

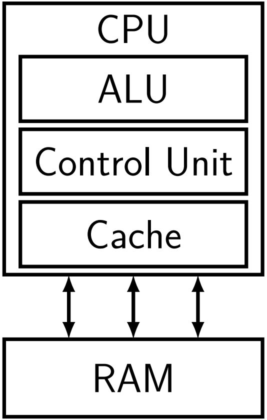
```
void laplace(double* u, double* x,
             double* y, double* f,
                  long Ns, long Nt) {
  for (long t = 0; t < Nt; t++) {
    for (long s = 0; s < Ns; s++) {
      double rx, ry, rz;
      rx = x[s*3]-y[t*3];
      ry = x[s*3+1]-y[t*3+1];
      rz = x[s*3+2]-y[t*3+2];

      double r2 = rx*rx+ry*ry+rz*rz;
      if (r2 > 0) {
        double rinv = 1/sqrt(r2);
        u[t] += f[s] * rinv;
      }
    }
  }
}
```



- code executes line-by-line
- sequentially and in order
- one scalar operation at a time
- one operation per clock cycle

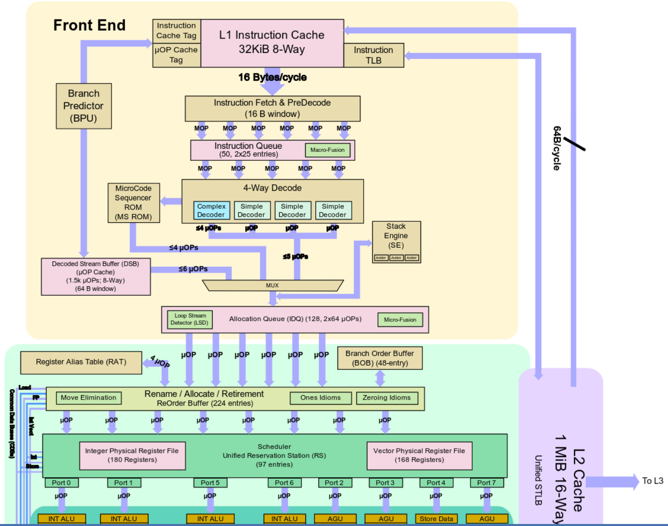
How code executes on a computer

```c
void laplace(double* u, double* x,
             double* y, double* f,
             long Ns, long Nt) {
  for (long t = 0; t < Nt; t++) {
    for (long s = 0; s < Ns; s++) {
      double rx, ry, rz;
      rx = x[s*3]-y[t*3];
      ry = x[s*3+1]-y[t*3+1];
      rz = x[s*3+2]-y[t*3+2];

      double r2 = rx*rx+ry*ry+rz*rz;
      if (r2 > 0) {
        double rinv = 1/sqrt(r2);
        u[t] += f[s] * rinv;
      }
    }
  }
}
```



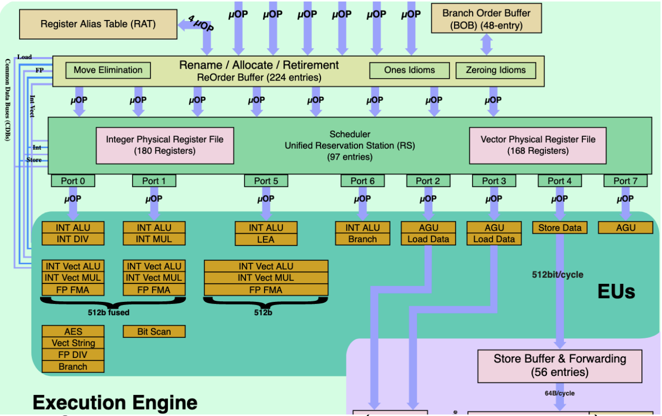
- code executes line-by-line
- sequentially and in order
- one scalar operation at a time
- one operation per clock cycle

Core microarchitecture



- Branch prediction and speculative execution
- Out-of-order execution

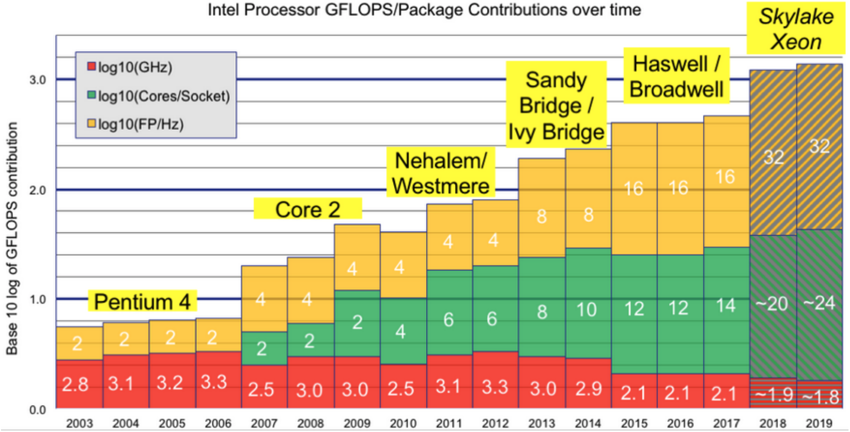
Core microarchitecture



Skylake micro-architecture (source: wikichip.org)

- Branch prediction and speculative execution
- Out-of-order execution
- Superscalar execution:
  2-FP, 2-reads, 1-write
- Vector instructions
- Pipelining: 'assembly line'
  latency and throughput

Instruction level parallelism



Source: John McCalpin - Memory bandwidth and system balance in HPC systems, 2016

Instruction latency and throughput

```cpp
#include <iostream>
#include <omp.h>

int main(int argc, char** argv) {
  double x = 3.141, one = 1.0;

  double T = -omp_get_wtime();
  for (long i = 0; i < 1000000000L; i++) {
    x = one + x;
  }
  T += omp_get_wtime();
  std::cout<<"T = "<< T <<'\n';
  std::cout<<"cycles/iter = "<< 3.3*T <<'\n';

  return 0;
}
```

```
$ g++ -O3 -march=native -fopenmp test.cpp
$ ./a.out
T = 0
cycles/iter = 0
```

FLATIRON INSTITUTE

Instruction latency and throughput

```
#include <iostream>
#include <omp.h>

int main(int argc, char** argv) {
  double x = 3.141, one = 1.0;

  double T = -omp_get_wtime();
  for (long i = 0; i < 1000000000L; i++) {
    x = one + x;
  }
  T += omp_get_wtime();
  std::cout<<"T = "<< T <<'\n';
  std::cout<<"cycles/iter = "<< 3.3*T <<'\n';

  std::cout<<x<<'\n';
  return 0;
}
```

```
$ g++ -O3 -march=native -fopenmp test.cpp
$ ./a.out
T = 0
cycles/iter = 0
```

FLATIRON INSTITUTE

Instruction latency and throughput

```cpp
#include <iostream>
#include <omp.h>

int main(int argc, char** argv) {
  double x = 3.141, one = 1.0;

  double T = -omp_get_wtime();
  for (long i = 0; i < 1000000000L; i++) {
    x = one + x;
  }
  T += omp_get_wtime();
  std::cout<<"T = "<< T <<'\n';
  std::cout<<"cycles/iter = "<< 3.3*T <<'\n';

  std::cout<<x<<'\n';
  return 0;
}
```

```
$ g++ -O3 -march=native -fopenmp test.cpp
$ ./a.out
T = 0
cycles/iter = 0
```

```
$ g++ -O3 -march=native -fopenmp test.cpp
$ ./a.out
T = 1.22387
cycles/iter = 4.03876
```

FLATIRON INSTITUTE

Instruction latency and throughput

```
double x[32], one = 1;
// ... initialize x

double T = -omp_get_wtime();
for (long i = 0; i < 1000000000L; i++) {
  x[0] = one + x[0];
  x[1] = one + x[1];
  x[2] = one + x[2];
  x[3] = one + x[3];
  ...
  x[31] = one + x[31];
}
T += omp_get_wtime();
std::cout<<"T = "<< T <<'\n';
std::cout<<"cycles/iter = "<< 3.3*T <<'\n';
```

```
$ g++ -O3 -march=native -fopenmp test.cpp
$ ./a.out
T = 0
cycles/iter = 0
```

```
$ g++ -O3 -march=native -fopenmp test.cpp
$ ./a.out
T = 1.22387
cycles/iter = 4.03876
```

FLATIRON INSTITUTE

Instruction latency and throughput

```
double x[32], one = 1;
// ... initialize x

double T = -omp_get_wtime();
for (long i = 0; i < 1000000000L; i++) {
  x[0] = one + x[0];
  x[1] = one + x[1];
  x[2] = one + x[2];
  x[3] = one + x[3];
  ...
  x[31] = one + x[31];
}
T += omp_get_wtime();
std::cout<<"T = "<< T <<'\n';
std::cout<<"cycles/iter = "<< 3.3*T <<'\n';
```

```
$ g++ -O3 -march=native -fopenmp test.cpp
$ ./a.out
T = 0
cycles/iter = 0
```

```
$ g++ -O3 -march=native -fopenmp test.cpp
$ ./a.out
T = 1.22387
cycles/iter = 4.03876
```

```
$ g++ -O3 -march=native -fopenmp test.cpp
$ ./a.out
T = 1.22366
cycles/iter = 4.03809
```

8 adds/cycle!

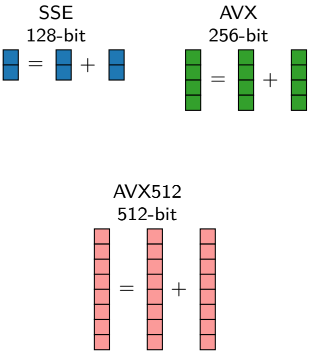
FLATIRON INSTITUTE

SIMD vector instructions

- Think in vectors instead of scalars (float, double)
- Re-organize computations as vector operations
  - Struct-of-arrays (SOA)

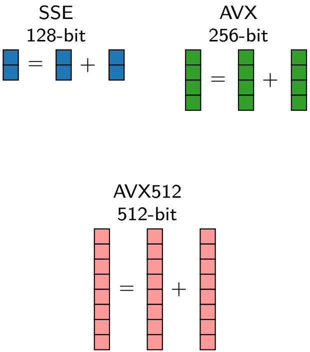
    $\{x_1, y_1, z_1, \quad x_2, y_2, z_2, \cdots, \quad x_n, y_n, z_n\}$
  - Array-of-struct (AOS)

    $\{x_1, \cdots, x_n, \quad y_1, \cdots, y_n, \quad z_1, \cdots, z_n\}$
- Tell the compiler it is safe to use SIMD instructions
  - most languages don't make it easy to specify when it is safe to vectorize (aliasing)



SIMD vector instructions

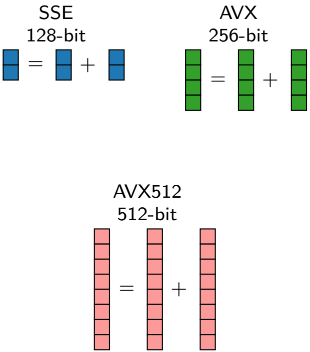
- Auto vectorization: unreliable!
  - Compiler specific hints:
    -fopt-info-vec-optimized
    __builtin_assume_aligned(a, 32)
    #pragma ivdep
  - OpenMP 4.0: #pragma omp simd
- Assembly: too hard!
- Vector intrinsics: works but messy!
  - _mm512_add_pd(__m512d, __m512d)
  - _mm512_mul_pd(__m512d, __m512d)
- C++ vector libraries: intuitive and clean



FLATIRON INSTITUTE

SIMD vector instructions

- C++ vector libraries: intuitive and clean
  - Vector objects, overloaded operators (+, -, *, ||, && etc)
  - Vector Class Library - Agner Fog
    `https://github.com/vectorclass/version2`
  - SLEEF Vectorized Math Library
  - SCTL (`https://github.com/dmalhotra/SCTL`)
  - Similar proposals for future C++ standard library
    `https://en.cppreference.com/w/cpp/experimental/simd`



Instruction latency and throughput

```
sctl::Vec<double,8> x[8], one = 1;
// ... initialize x

double T = -omp_get_wtime();
for (long i = 0; i < 1000000000L; i++) {
  x[0] = one + x[0];
  x[1] = one + x[1];
  x[2] = one + x[2];
  x[3] = one + x[3];
  ...
  x[8] = one + x[8];
}
T += omp_get_wtime();
std::cout<<"T = "<< T <<'\n';
std::cout<<"cycles/iter = "<< 3.3*T <<'\n';
```

FLATIRON INSTITUTE

Instruction latency and throughput

```
sctl::Vec<double,8> x[8], one = 1;
// ... initialize x

double T = -omp_get_wtime();
for (long i = 0; i < 1000000000L; i++) {
  x[0] = one + x[0];
  x[1] = one + x[1];
  x[2] = one + x[2];
  x[3] = one + x[3];
  ...
  x[8] = one + x[8];
}
T += omp_get_wtime();
std::cout<<"T = "<< T <<'\n';
std::cout<<"cycles/iter = "<< 3.3*T <<'\n';
```

```
T = 1.22806
cycles/iter = 4.05259
```

16 adds/cycle!

FLATIRON INSTITUTE

Instruction latency and throughput

```
sctl::Vec<double,8> x[8], one = 1;
// ... initialize x

double T = -omp_get_wtime();
for (long i = 0; i < 1000000000L; i++) {
  x[0] = one / x[0];
  x[1] = one / x[1];
  x[2] = one / x[2];
  x[3] = one / x[3];
  ...
  x[8] = one / x[8];
}
T += omp_get_wtime();
std::cout<<"T = "<< T <<'\n';
std::cout<<"cycles/iter = "<< 3.3*T <<'\n';
```

```
T = 1.22806
cycles/iter = 4.05259
```

16 adds/cycle!

— floating-point division —

Instruction latency and throughput

```
sctl::Vec<double,8> x[8], one = 1;
// ... initialize x

double T = -omp_get_wtime();
for (long i = 0; i < 1000000000L; i++) {
  x[0] = one / x[0];
  x[1] = one / x[1];
  x[2] = one / x[2];
  x[3] = one / x[3];
  ...
  x[8] = one / x[8];
}
T += omp_get_wtime();
std::cout<<"T = "<< T <<'\n';
std::cout<<"cycles/iter = "<< 3.3*T <<'\n';
```

```
T = 1.22806
cycles/iter = 4.05259
```

16 adds/cycle!

— floating-point division —

```
T = 39.1521
cycles/iter = 129.202
```

$\sim 32\times$ slower!

FLATIRON INSTITUTE

Instruction latency and throughput

```
sctl::Vec<double,8> x[8], one = 1;
// ... initialize x

double T = -omp_get_wtime();
for (long i = 0; i < 1000000000L; i++) {
  x[0] = one / x[0];
  x[1] = one / x[1];
  x[2] = one / x[2];
  x[3] = one / x[3];
  ...
  x[8] = one / x[8];
}
T += omp_get_wtime();
std::cout<<"T = "<< T <<'\n';
std::cout<<"cycles/iter = "<< 3.3*T <<'\n';
```

```
T = 1.22806
cycles/iter = 4.05259
```

16 adds/cycle!

— floating-point division —

```
T = 39.1521
cycles/iter = 129.202
```

$\sim 32\times$ slower!

Fast: bitwise ops, int & fp ops $(+, -, *)$

Slow: branches, $/, \sqrt{\cdot}, \sin, \cos, \cdots$

FLATIRON INSTITUTE

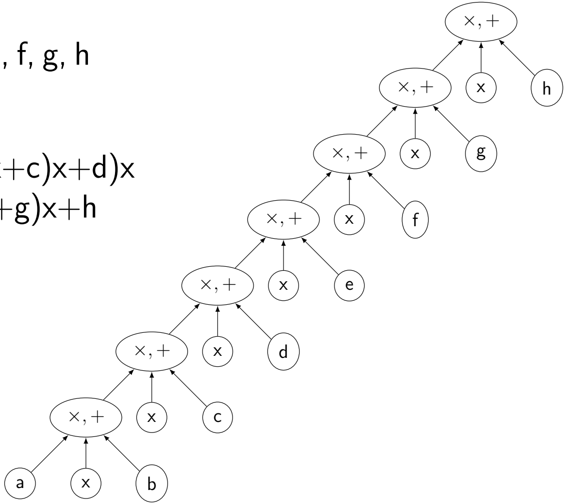
Pipelining polynomial eval (Horner's rule)

Input:

x, a, b, c, d, e, f, g, h

Compute:

$((((((ax+b)x+c)x+d)x+e)x+f)x+g)x+h$



$u = a * x + b \leftarrow$

$v = u * x + c$

$w = v * x + d$

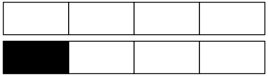
$p = w * x + e$

$q = p * x + f$

$r = q * x + g$

$s = r * x + h$

Pipeline:



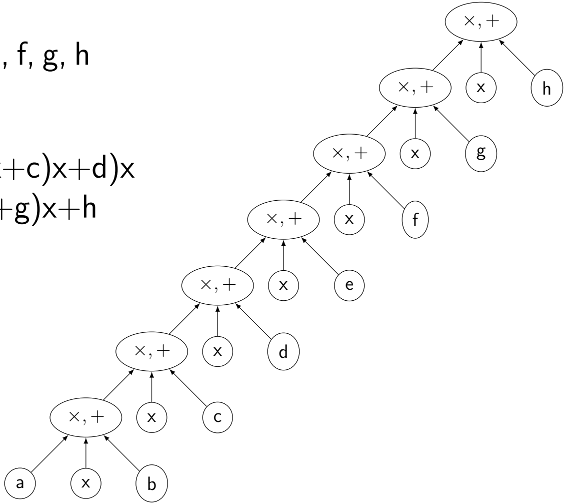
Pipelining polynomial eval (Horner's rule)

Input:

x, a, b, c, d, e, f, g, h

Compute:

$((((((ax+b)x+c)x+d)x$
$+e)x+f)x+g)x+h$



$u = a * x + b \leftarrow$

$v = u * x + c$

$w = v * x + d$

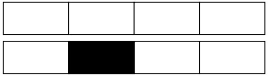
$p = w * x + e$

$q = p * x + f$

$r = q * x + g$

$s = r * x + h$

Pipeline:



FLATIRON INSTITUTE

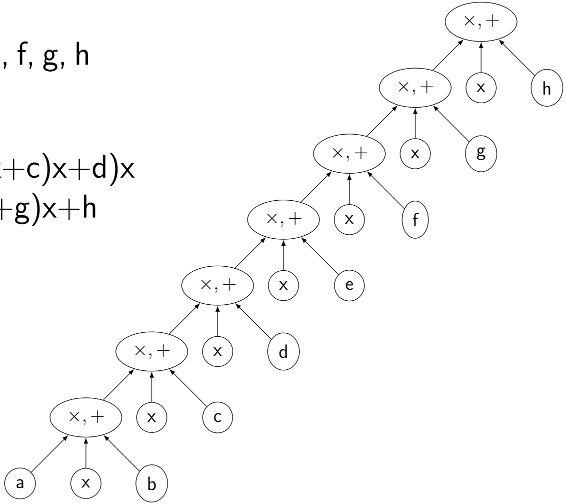
Pipelining polynomial eval (Horner's rule)

Input:

x, a, b, c, d, e, f, g, h

Compute:

$((((((ax+b)x+c)x+d)x$
$\quad +e)x+f)x+g)x+h$



$u = a * x + b \leftarrow$

$v = u * x + c$

$w = v * x + d$

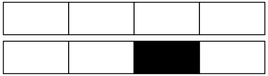
$p = w * x + e$

$q = p * x + f$

$r = q * x + g$

$s = r * x + h$

Pipeline:



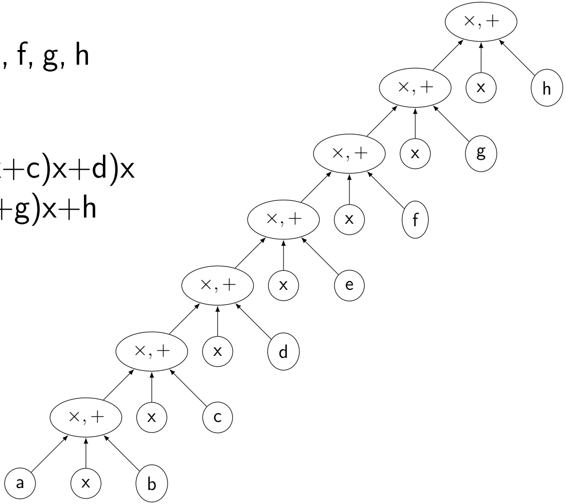
Pipelining polynomial eval (Horner's rule)

Input:

x, a, b, c, d, e, f, g, h

Compute:

(((((((ax+b)x+c)x+d)x
+e)x+f)x+g)x+h



$u = a * x + b \leftarrow$

$v = u * x + c$

$w = v * x + d$

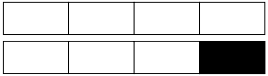
$p = w * x + e$

$q = p * x + f$

$r = q * x + g$

$s = r * x + h$

Pipeline:



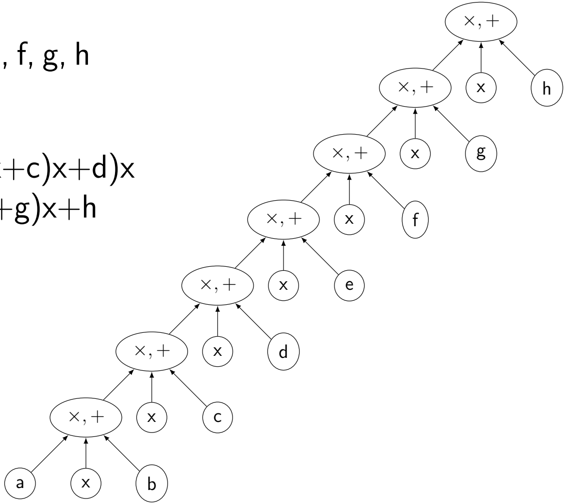
Pipelining polynomial eval (Horner's rule)

Input:

x, a, b, c, d, e, f, g, h

Compute:

$$((((((ax+b)x+c)x+d)x+e)x+f)x+g)x+h$$



$u = a * x + b$

$v = u * x + c \leftarrow$

$w = v * x + d$

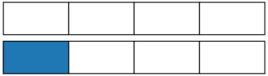
$p = w * x + e$

$q = p * x + f$

$r = q * x + g$

$s = r * x + h$

Pipeline:



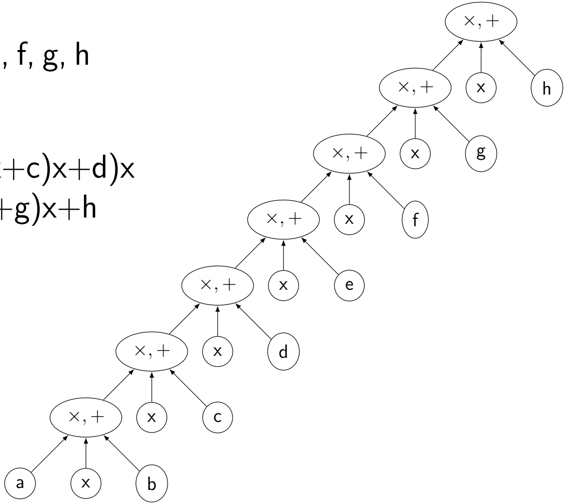
Pipelining polynomial eval (Horner's rule)

Input:

x, a, b, c, d, e, f, g, h

Compute:

$((((((ax+b)x+c)x+d)x$
$+e)x+f)x+g)x+h$



$u = a * x + b$

$v = u * x + c \leftarrow$

$w = v * x + d$

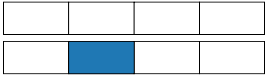
$p = w * x + e$

$q = p * x + f$

$r = q * x + g$

$s = r * x + h$

Pipeline:



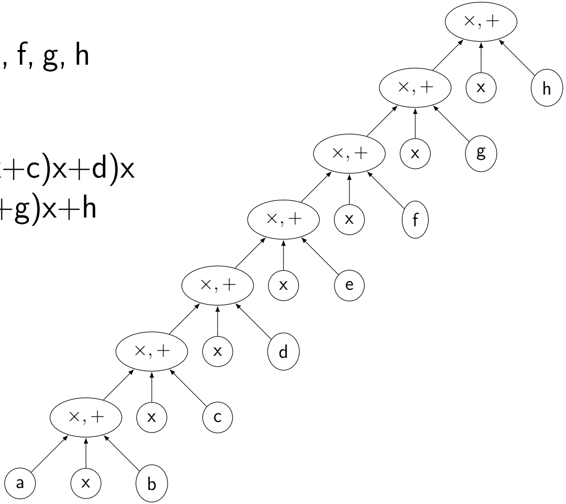
Pipelining polynomial eval (Horner's rule)

Input:

x, a, b, c, d, e, f, g, h

Compute:

$((((((ax+b)x+c)x+d)x$
$\quad +e)x+f)x+g)x+h$



$u = a * x + b$

$v = u * x + c \leftarrow$

$w = v * x + d$

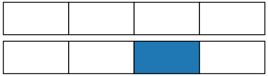
$p = w * x + e$

$q = p * x + f$

$r = q * x + g$

$s = r * x + h$

Pipeline:



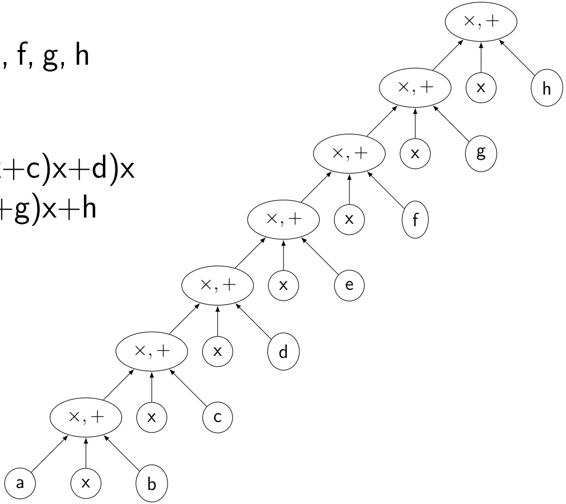
Pipelining polynomial eval (Horner's rule)

Input:

x, a, b, c, d, e, f, g, h

Compute:

(((((((ax+b)x+c)x+d)x
+e)x+f)x+g)x+h



$u = a * x + b$

$v = u * x + c \leftarrow$

$w = v * x + d$

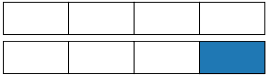
$p = w * x + e$

$q = p * x + f$

$r = q * x + g$

$s = r * x + h$

Pipeline:



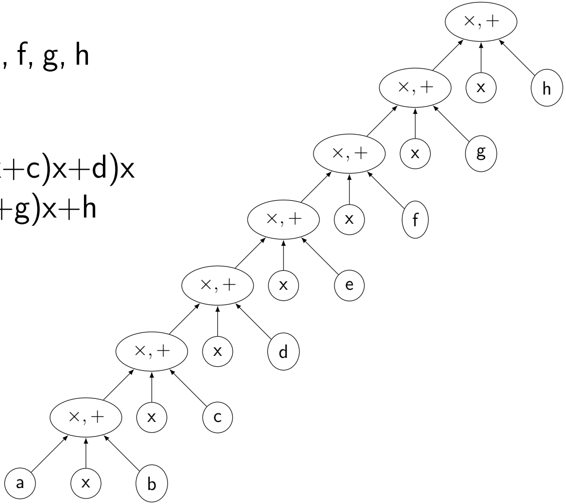
Pipeline polynomial eval (Horner's rule)

Input:

x, a, b, c, d, e, f, g, h

Compute:

$$((((((ax+b)x+c)x+d)x+e)x+f)x+g)x+h$$



u = a * x + b

v = u * x + c

w = v * x + d ←

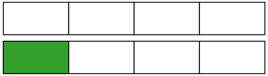
p = w * x + e

q = p * x + f

r = q * x + g

s = r * x + h

Pipeline:



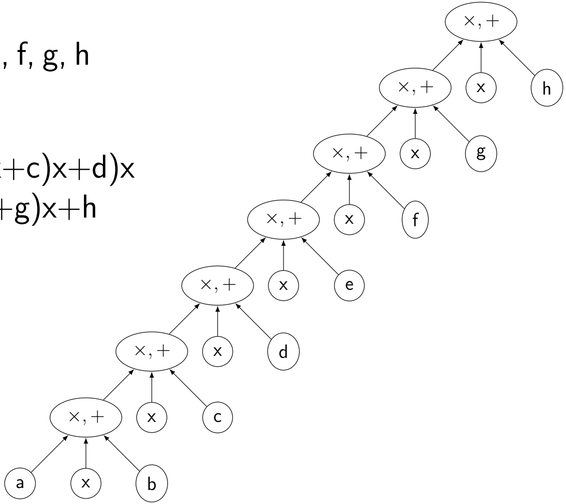
Pipelining polynomial eval (Horner's rule)

Input:

x, a, b, c, d, e, f, g, h

Compute:

(((((((ax+b)x+c)x+d)x
+e)x+f)x+g)x+h



$u = a * x + b$

$v = u * x + c$

$w = v * x + d \leftarrow$

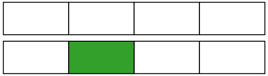
$p = w * x + e$

$q = p * x + f$

$r = q * x + g$

$s = r * x + h$

Pipeline:



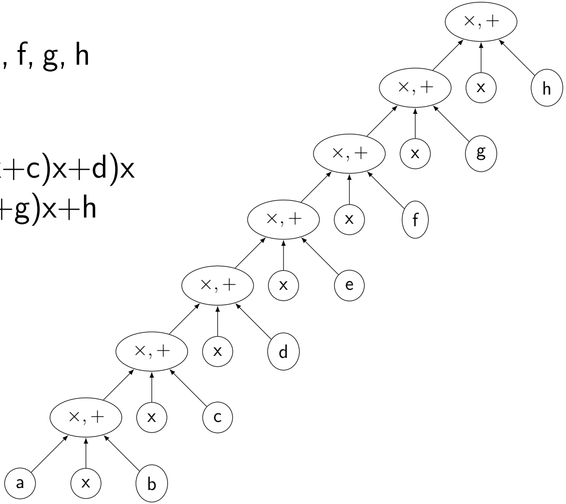
Pipelining polynomial eval (Horner's rule)

Input:

x, a, b, c, d, e, f, g, h

Compute:

$((((((ax+b)x+c)x+d)x$
$\quad +e)x+f)x+g)x+h$



$u = a * x + b$

$v = u * x + c$

$w = v * x + d \leftarrow$

$p = w * x + e$

$q = p * x + f$

$r = q * x + g$

$s = r * x + h$

Pipeline:

FLATIRON INSTITUTE

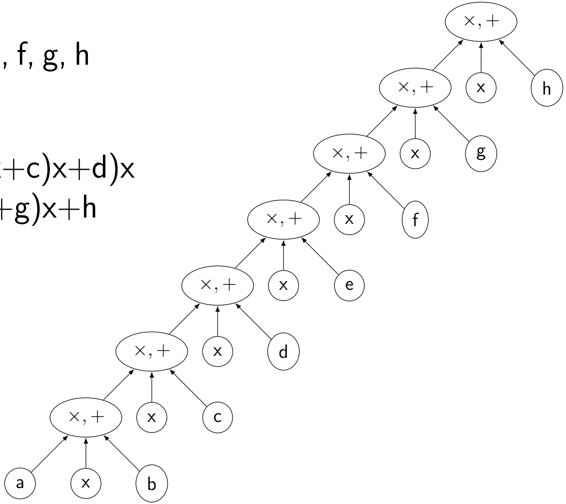
Pipelining polynomial eval (Horner's rule)

Input:

x, a, b, c, d, e, f, g, h

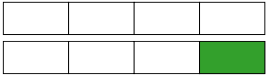
Compute:

$(((((((ax+b)x+c)x+d)x+e)x+f)x+g)x+h$



u = a * x + b
v = u * x + c
w = v * x + d ←
p = w * x + e
q = p * x + f
r = q * x + g
s = r * x + h

Pipeline:



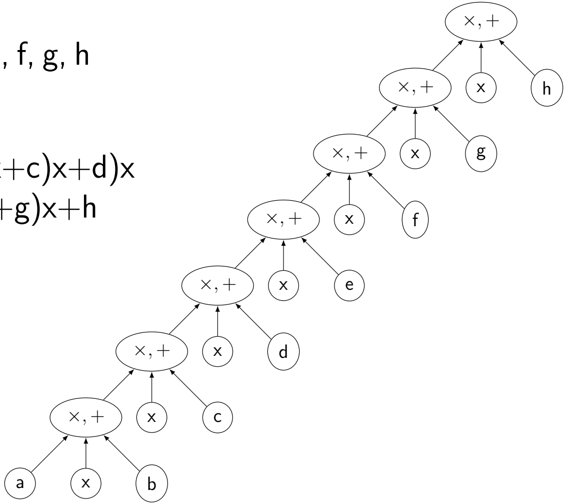
Pipelining polynomial eval (Horner's rule)

Input:

x, a, b, c, d, e, f, g, h

Compute:

$$((((((ax+b)x+c)x+d)x+e)x+f)x+g)x+h$$



$u = a * x + b$

$v = u * x + c$

$w = v * x + d$

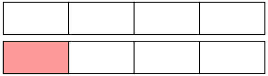
$p = w * x + e$ ←

$q = p * x + f$

$r = q * x + g$

$s = r * x + h$

Pipeline:



FLATIRON INSTITUTE

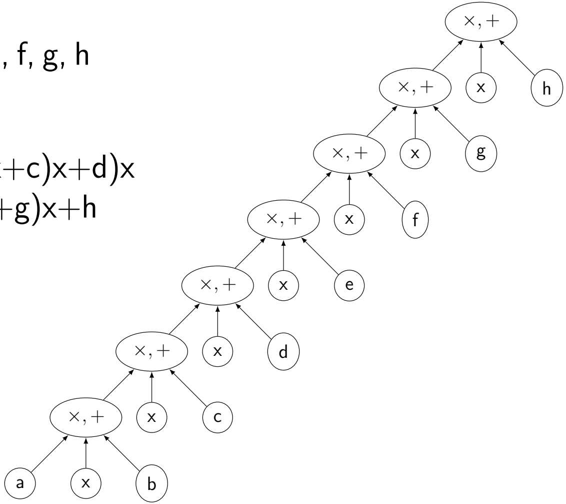
Pipelining polynomial eval (Horner's rule)

Input:

x, a, b, c, d, e, f, g, h

Compute:

$$((((((ax+b)x+c)x+d)x+e)x+f)x+g)x+h$$



$u = a * x + b$

$v = u * x + c$

$w = v * x + d$

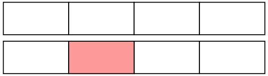
$p = w * x + e \leftarrow$

$q = p * x + f$

$r = q * x + g$

$s = r * x + h$

Pipeline:



FLATIRON INSTITUTE

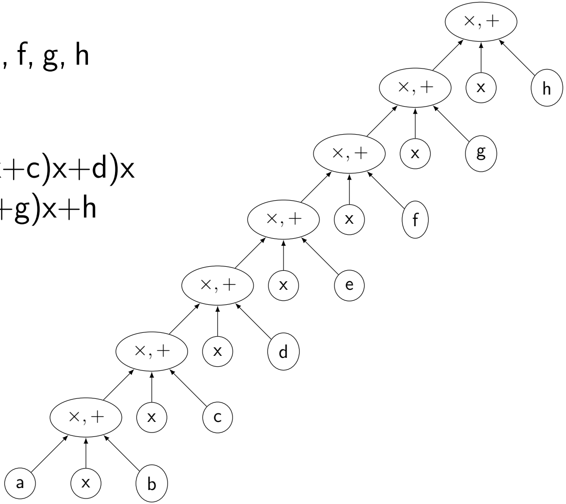
Pipelining polynomial eval (Horner's rule)

Input:

x, a, b, c, d, e, f, g, h

Compute:

$$((((((ax+b)x+c)x+d)x \\ +e)x+f)x+g)x+h$$



$u = a * x + b$

$v = u * x + c$

$w = v * x + d$

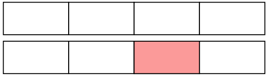
$p = w * x + e$ ←

$q = p * x + f$

$r = q * x + g$

$s = r * x + h$

Pipeline:



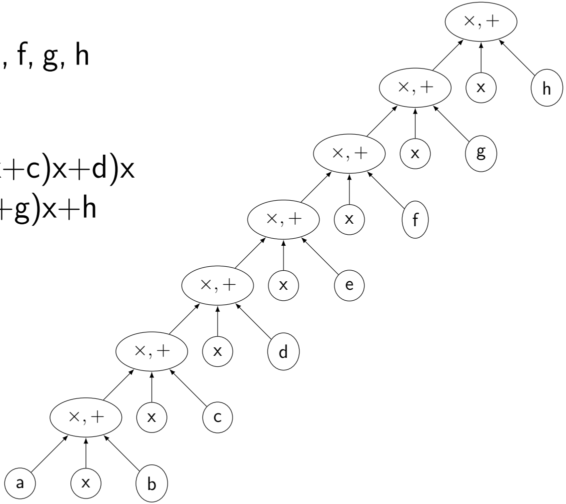
Pipelining polynomial eval (Horner's rule)

Input:

x, a, b, c, d, e, f, g, h

Compute:

$$((((((ax+b)x+c)x+d)x+e)x+f)x+g)x+h$$



$u = a * x + b$

$v = u * x + c$

$w = v * x + d$

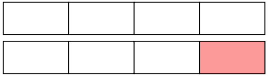
$p = w * x + e$ ←

$q = p * x + f$

$r = q * x + g$

$s = r * x + h$

Pipeline:



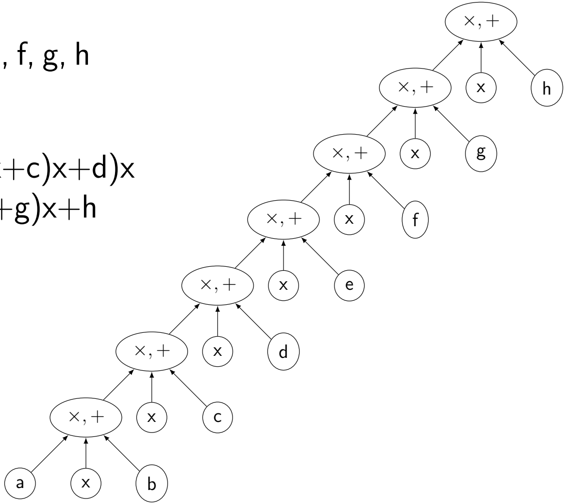
Pipelining polynomial eval (Horner's rule)

Input:

x, a, b, c, d, e, f, g, h

Compute:

$((((((ax+b)x+c)x+d)x$
$+e)x+f)x+g)x+h$



$u = a * x + b$

$v = u * x + c$

$w = v * x + d$

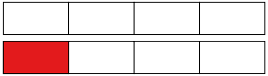
$p = w * x + e$

$q = p * x + f \leftarrow$

$r = q * x + g$

$s = r * x + h$

Pipeline:



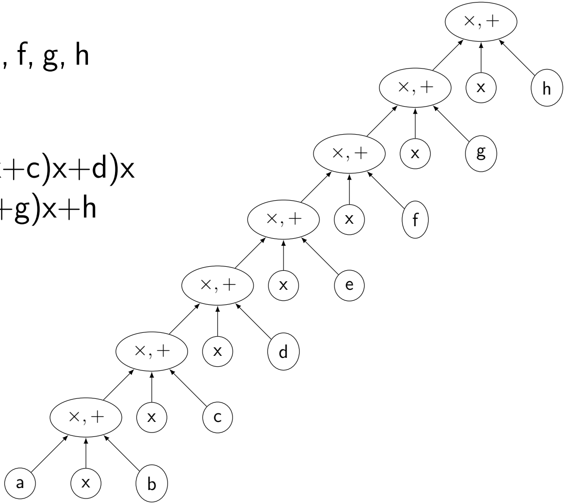
Pipelining polynomial eval (Horner's rule)

Input:

x, a, b, c, d, e, f, g, h

Compute:

$((((((ax+b)x+c)x+d)x + e)x+f)x+g)x+h$



$u = a * x + b$

$v = u * x + c$

$w = v * x + d$

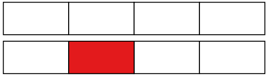
$p = w * x + e$

$q = p * x + f \leftarrow$

$r = q * x + g$

$s = r * x + h$

Pipeline:



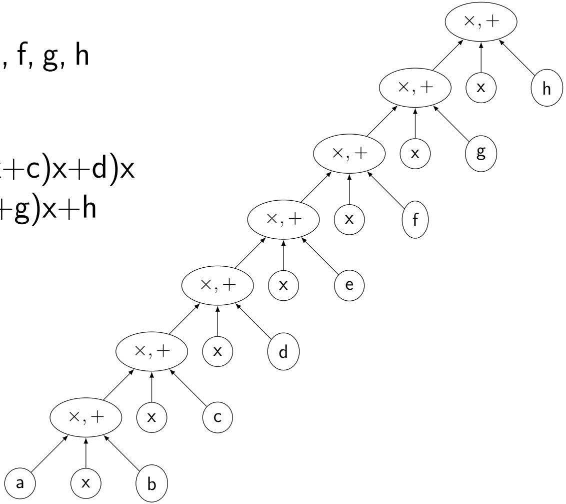
Pipelining polynomial eval (Horner's rule)

Input:

x, a, b, c, d, e, f, g, h

Compute:

$((((((ax+b)x+c)x+d)x$
$+e)x+f)x+g)x+h$



$u = a * x + b$

$v = u * x + c$

$w = v * x + d$

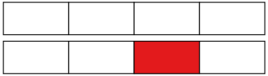
$p = w * x + e$

$q = p * x + f \leftarrow$

$r = q * x + g$

$s = r * x + h$

Pipeline:



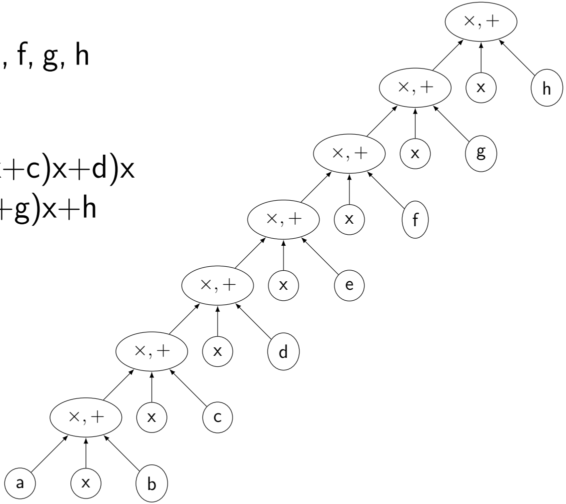
Pipelining polynomial eval (Horner's rule)

Input:

x, a, b, c, d, e, f, g, h

Compute:

$((((((ax+b)x+c)x+d)x+e)x+f)x+g)x+h$



$u = a * x + b$

$v = u * x + c$

$w = v * x + d$

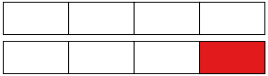
$p = w * x + e$

$q = p * x + f \leftarrow$

$r = q * x + g$

$s = r * x + h$

Pipeline:



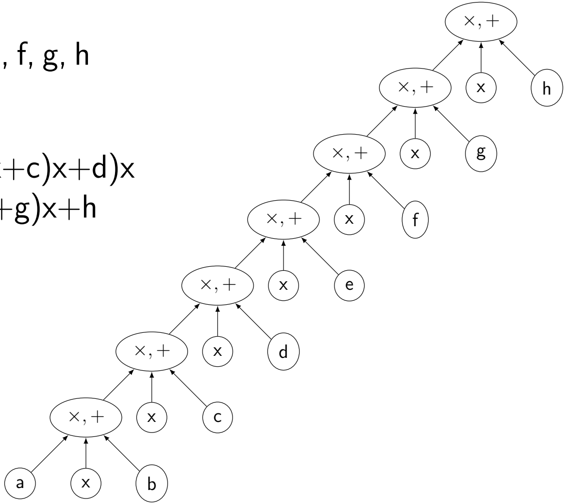
Pipelining polynomial eval (Horner's rule)

Input:

x, a, b, c, d, e, f, g, h

Compute:

$((((((ax+b)x+c)x+d)x$
$+e)x+f)x+g)x+h$



$u = a * x + b$

$v = u * x + c$

$w = v * x + d$

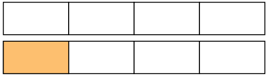
$p = w * x + e$

$q = p * x + f$

$r = q * x + g \leftarrow$

$s = r * x + h$

Pipeline:



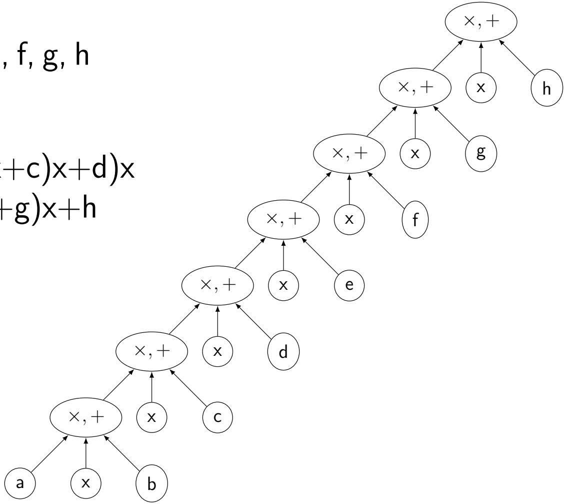
Pipelining polynomial eval (Horner's rule)

Input:

x, a, b, c, d, e, f, g, h

Compute:

$(((((((ax+b)x+c)x+d)x$
$+e)x+f)x+g)x+h$



$u = a * x + b$

$v = u * x + c$

$w = v * x + d$

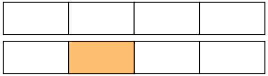
$p = w * x + e$

$q = p * x + f$

$r = q * x + g \leftarrow$

$s = r * x + h$

Pipeline:



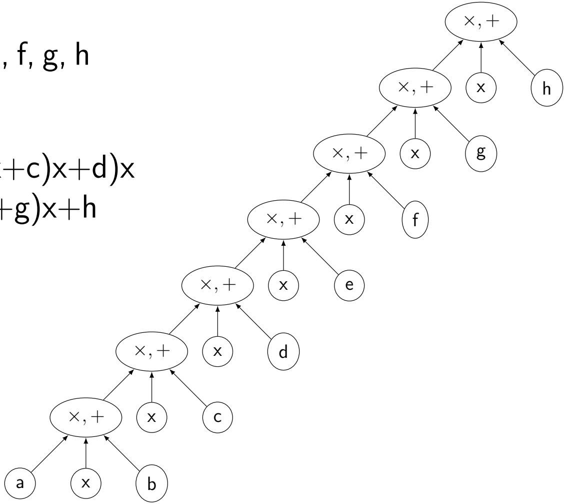
Pipeline polynomial eval (Horner's rule)

Input:

x, a, b, c, d, e, f, g, h

Compute:

$((((((ax+b)x+c)x+d)x$
$+e)x+f)x+g)x+h$



$u = a * x + b$

$v = u * x + c$

$w = v * x + d$

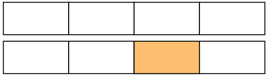
$p = w * x + e$

$q = p * x + f$

$r = q * x + g \leftarrow$

$s = r * x + h$

Pipeline:



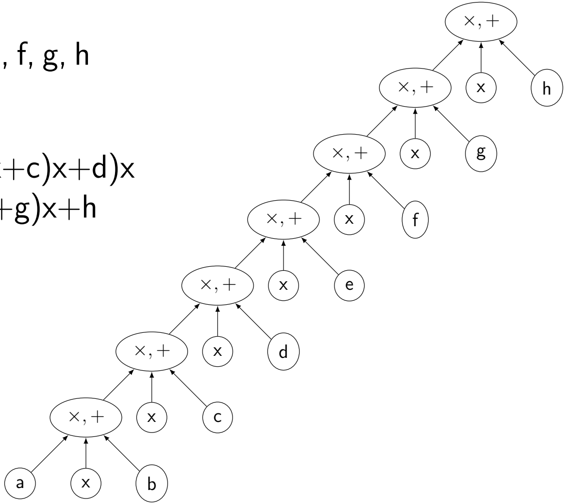
Pipelining polynomial eval (Horner's rule)

Input:

x, a, b, c, d, e, f, g, h

Compute:

$$((((((ax+b)x+c)x+d)x+e)x+f)x+g)x+h$$



$u = a * x + b$

$v = u * x + c$

$w = v * x + d$

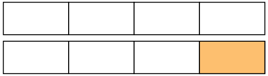
$p = w * x + e$

$q = p * x + f$

$r = q * x + g \leftarrow$

$s = r * x + h$

Pipeline:



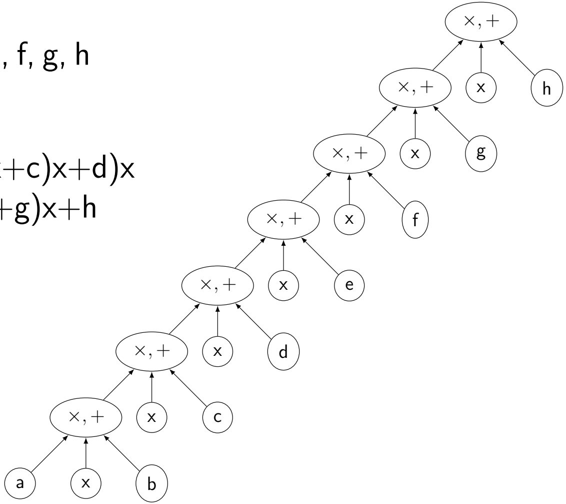
Pipelining polynomial eval (Horner's rule)

Input:

x, a, b, c, d, e, f, g, h

Compute:

$(((((((ax+b)x+c)x+d)x$
$\quad +e)x+f)x+g)x+h$



$u = a * x + b$

$v = u * x + c$

$w = v * x + d$

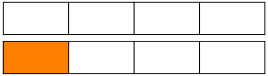
$p = w * x + e$

$q = p * x + f$

$r = q * x + g$

$s = r * x + h \leftarrow$

Pipeline:



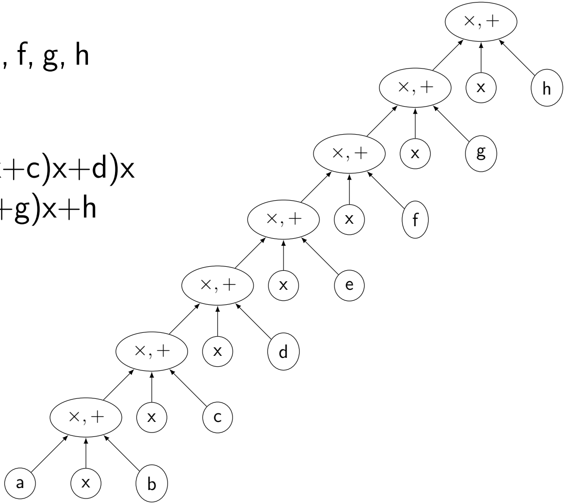
Pipelining polynomial eval (Horner's rule)

Input:

x, a, b, c, d, e, f, g, h

Compute:

$((((((ax+b)x+c)x+d)x$
$\quad +e)x+f)x+g)x+h$



$u = a * x + b$

$v = u * x + c$

$w = v * x + d$

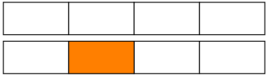
$p = w * x + e$

$q = p * x + f$

$r = q * x + g$

$s = r * x + h \leftarrow$

Pipeline:



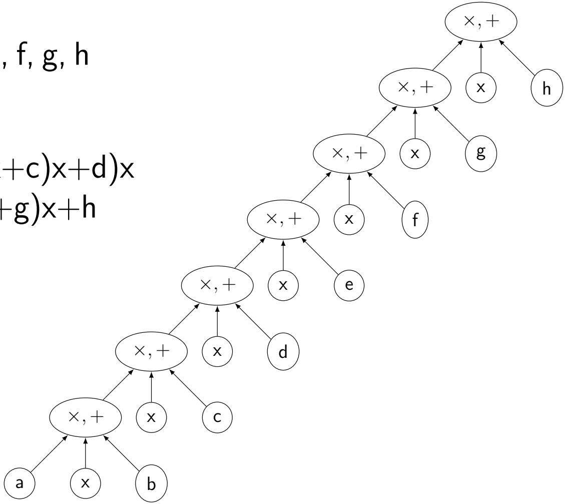
Pipelining polynomial eval (Horner's rule)

Input:

x, a, b, c, d, e, f, g, h

Compute:

$((((((ax+b)x+c)x+d)x$
$+e)x+f)x+g)x+h$



$u = a * x + b$

$v = u * x + c$

$w = v * x + d$

$p = w * x + e$

$q = p * x + f$

$r = q * x + g$

$s = r * x + h \leftarrow$

Pipeline:

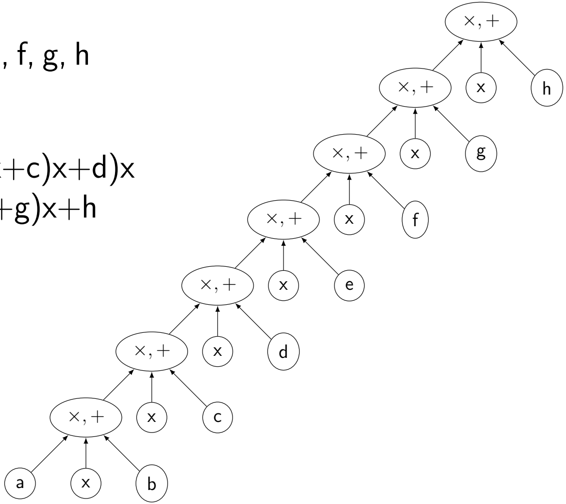
Pipelining polynomial eval (Horner's rule)

Input:

x, a, b, c, d, e, f, g, h

Compute:

$(((((((ax+b)x+c)x+d)x$
$+e)x+f)x+g)x+h$



$u = a * x + b$

$v = u * x + c$

$w = v * x + d$

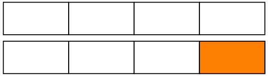
$p = w * x + e$

$q = p * x + f$

$r = q * x + g$

$s = r * x + h \leftarrow$

Pipeline:



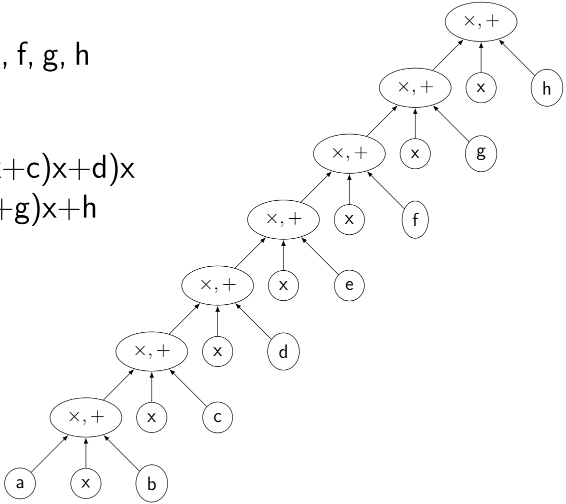
Pipelining polynomial eval (Horner's rule)

Input:

x, a, b, c, d, e, f, g, h

Compute:

$((((((ax+b)x+c)x+d)x+e)x+f)x+g)x+h$



$u = a * x + b$
$v = u * x + c$
$w = v * x + d$
$p = w * x + e$
$q = p * x + f$
$r = q * x + g$
$s = r * x + h$

Pipeline:

28 cycles
12.5% utilization!

FLATIRON INSTITUTE

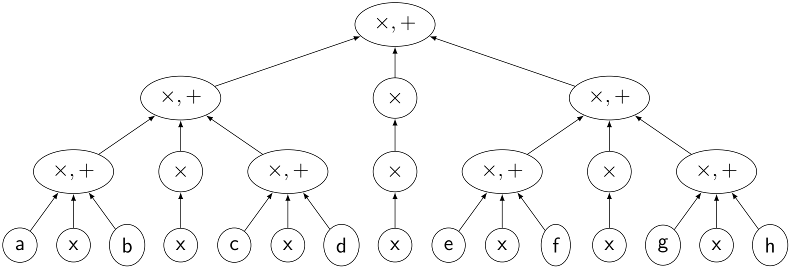
Pipelining: polynomial eval (Estrin's method)

Input:

x, a, b, c, d, e, f, g, h

Compute:

$((ax+b)x^2+(cx+d))x^4+(ex+f)x^2+(gx+h)$



$x^2 = x * x \leftarrow$

$x^4 = x^2 * x^2$

$u = a * x + b \leftarrow$

$v = c * x + d$

$w = e * x + f$

$p = g * x + h$

$q = u * x^2 + v$

$r = w * x^2 + p$

$s = q * x^4 + r$

Pipeline:

FLATIRON INSTITUTE

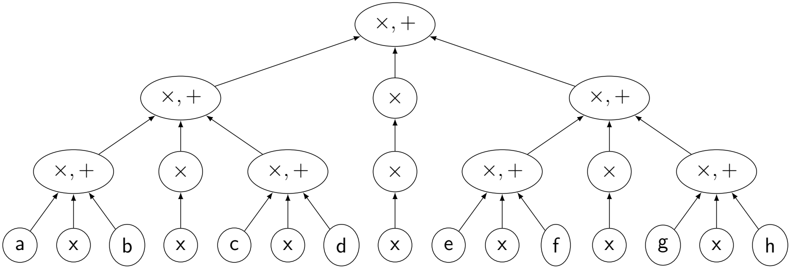
Pipelining: polynomial eval (Estrin's method)

Input:

x, a, b, c, d, e, f, g, h

Compute:

$((ax+b)x^2+(cx+d))x^4+(ex+f)x^2+(gx+h)$



$x^2 = x * x \leftarrow$

$x^4 = x^2 * x^2$

$u = a * x + b \leftarrow$

$v = c * x + d \leftarrow$

$w = e * x + f \leftarrow$

$p = g * x + h$

$q = u * x^2 + v$

$r = w * x^2 + p$

$s = q * x^4 + r$

Pipeline:

FLATIRON INSTITUTE

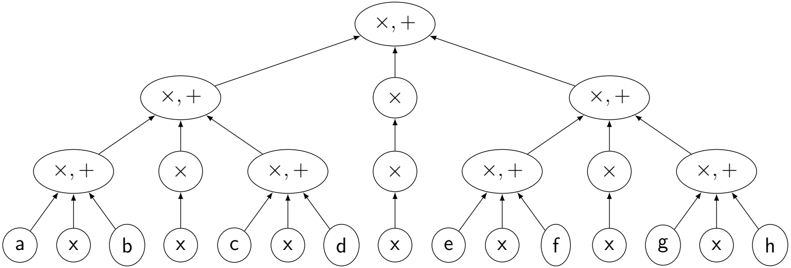
Pipelining: polynomial eval (Estrin's method)

Input:

x, a, b, c, d, e, f, g, h

Compute:

$((ax+b)x^2+(cx+d))x^4+(ex+f)x^2+(gx+h)$



$x^2 = x * x \leftarrow$

$x^4 = x^2 * x^2$

$u = a * x + b \leftarrow$

$v = c * x + d \leftarrow$

$w = e * x + f \leftarrow$

$p = g * x + h \leftarrow$

$q = u * x^2 + v$

$r = w * x^2 + p$

$s = q * x^4 + r$

Pipeline:

FLATIRON INSTITUTE

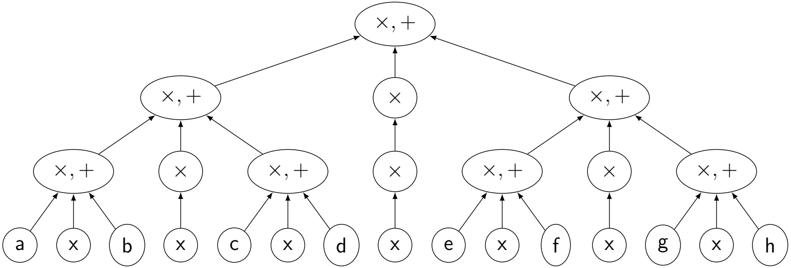
Pipelining: polynomial eval (Estrin's method)

Input:

x, a, b, c, d, e, f, g, h

Compute:

$((ax+b)x^2+(cx+d))x^4+(ex+f)x^2+(gx+h)$



$x^2 = x * x \leftarrow$

$x^4 = x^2 * x^2$

$u = a * x + b \leftarrow$

$v = c * x + d \leftarrow$

$w = e * x + f \leftarrow$

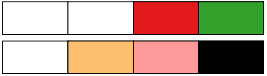
$p = g * x + h \leftarrow$

$q = u * x^2 + v$

$r = w * x^2 + p$

$s = q * x^4 + r$

Pipeline:



FLATIRON INSTITUTE

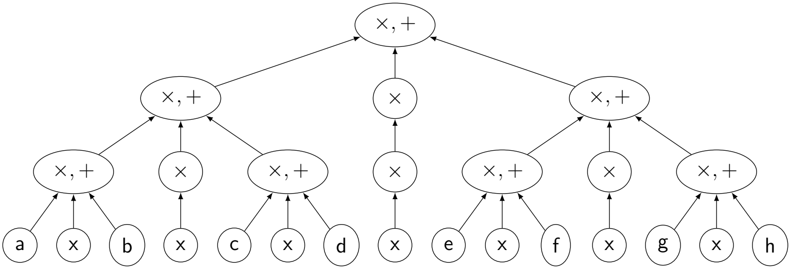
Pipelining: polynomial eval (Estrin's method)

Input:

x, a, b, c, d, e, f, g, h

Compute:

$((ax+b)x^2+(cx+d))x^4+(ex+f)x^2+(gx+h)$



$x^2 = x * x$

$x^4 = x^2 * x^2 \leftarrow$

$u = a * x + b$

$v = c * x + d \leftarrow$

$w = e * x + f \leftarrow$

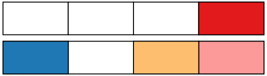
$p = g * x + h \leftarrow$

$q = u * x^2 + v$

$r = w * x^2 + p$

$s = q * x^4 + r$

Pipeline:



FLATIRON INSTITUTE

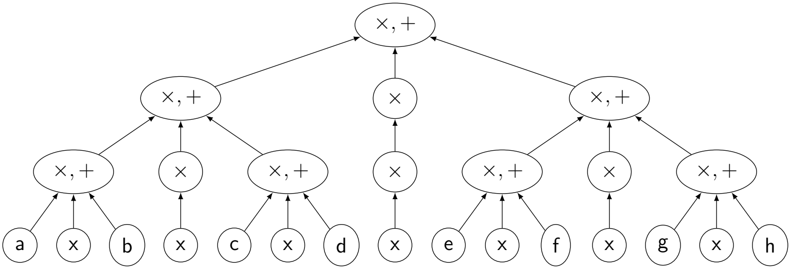
Pipelining: polynomial eval (Estrin's method)

Input:

x, a, b, c, d, e, f, g, h

Compute:

$((ax+b)x^2+(cx+d))x^4+(ex+f)x^2+(gx+h)$



$x^2 = x * x$

$x^4 = x^2 * x^2 \leftarrow$

$u = a * x + b$

$v = c * x + d$

$w = e * x + f$

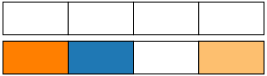
$p = g * x + h \leftarrow$

$q = u * x^2 + v \leftarrow$

$r = w * x^2 + p$

$s = q * x^4 + r$

Pipeline:



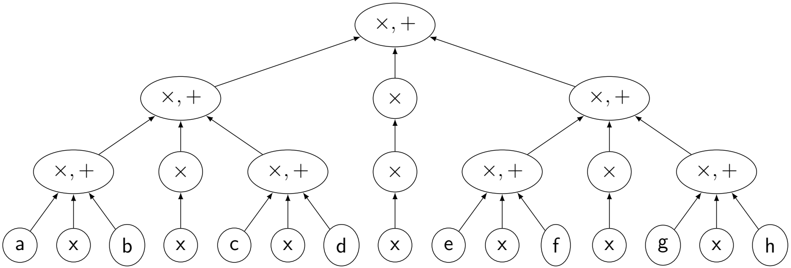
Pipelining: polynomial eval (Estrin's method)

Input:

x, a, b, c, d, e, f, g, h

Compute:

$((ax+b)x^2+(cx+d))x^4+(ex+f)x^2+(gx+h)$



$x^2 = x * x$

$x^4 = x^2 * x^2 \leftarrow$

$u = a * x + b$

$v = c * x + d$

$w = e * x + f$

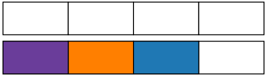
$p = g * x + h$

$q = u * x^2 + v \leftarrow$

$r = w * x^2 + p \leftarrow$

$s = q * x^4 + r$

Pipeline:



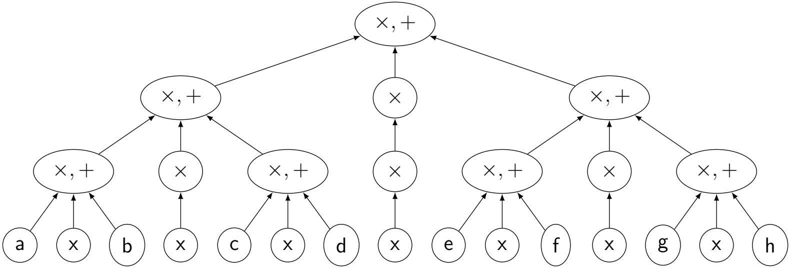
Pipelining: polynomial eval (Estrin's method)

Input:

x, a, b, c, d, e, f, g, h

Compute:

$((ax+b)x^2+(cx+d))x^4+(ex+f)x^2+(gx+h)$



$x^2 = x * x$

$x^4 = x^2 * x^2 \leftarrow$

$u = a * x + b$

$v = c * x + d$

$w = e * x + f$

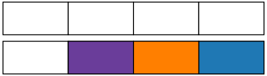
$p = g * x + h$

$q = u * x^2 + v \leftarrow$

$r = w * x^2 + p \leftarrow$

$s = q * x^4 + r$

Pipeline:



FLATIRON INSTITUTE

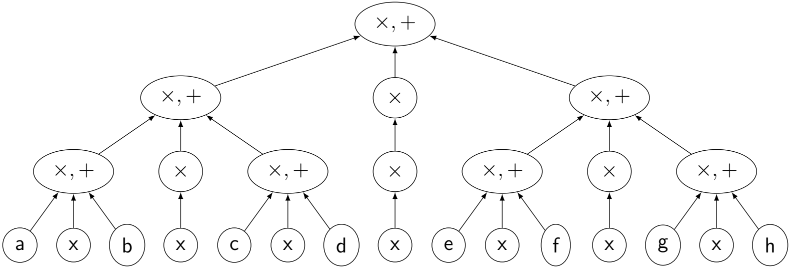
Pipelining: polynomial eval (Estrin's method)

Input:

x, a, b, c, d, e, f, g, h

Compute:

$((ax+b)x^2+(cx+d))x^4+(ex+f)x^2+(gx+h)$



$x^2 = x * x$

$x^4 = x^2 * x^2$

$u = a * x + b$

$v = c * x + d$

$w = e * x + f$

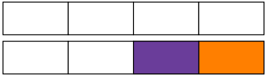
$p = g * x + h$

$q = u * x^2 + v \leftarrow$

$r = w * x^2 + p \leftarrow$

$s = q * x^4 + r$

Pipeline:



FLATIRON INSTITUTE

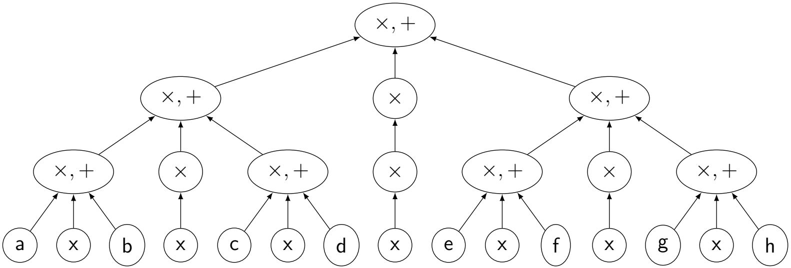
Pipelining: polynomial eval (Estrin's method)

Input:

x, a, b, c, d, e, f, g, h

Compute:

$((ax+b)x^2+(cx+d))x^4+(ex+f)x^2+(gx+h)$



$x^2 = x * x$

$x^4 = x^2 * x^2$

$u = a * x + b$

$v = c * x + d$

$w = e * x + f$

$p = g * x + h$

$q = u * x^2 + v$

$r = w * x^2 + p \leftarrow$

$s = q * x^4 + r$

Pipeline:

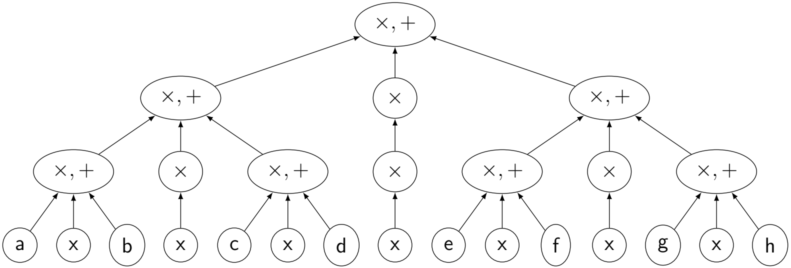
Pipelining: polynomial eval (Estrin's method)

Input:

x, a, b, c, d, e, f, g, h

Compute:

$((ax+b)x^2+(cx+d))x^4+(ex+f)x^2+(gx+h)$



$x^2 = x * x$

$x^4 = x^2 * x^2$

$u = a * x + b$

$v = c * x + d$

$w = e * x + f$

$p = g * x + h$

$q = u * x^2 + v$

$r = w * x^2 + p$

$s = q * x^4 + r \leftarrow$

Pipeline:

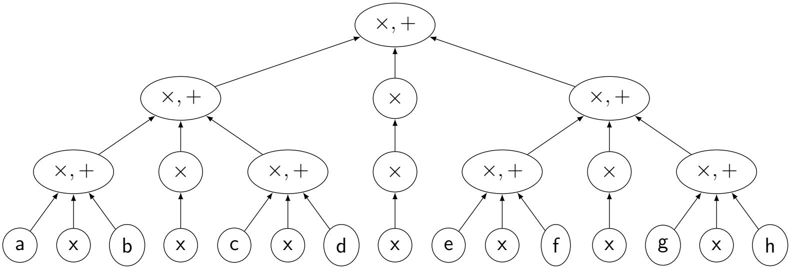
Pipelining: polynomial eval (Estrin's method)

Input:

x, a, b, c, d, e, f, g, h

Compute:

$((ax+b)x^2+(cx+d))x^4+(ex+f)x^2+(gx+h)$



$x^2 = x * x$

$x^4 = x^2 * x^2$

$u = a * x + b$

$v = c * x + d$

$w = e * x + f$

$p = g * x + h$

$q = u * x^2 + v$

$r = w * x^2 + p$

$s = q * x^4 + r \leftarrow$

Pipeline:

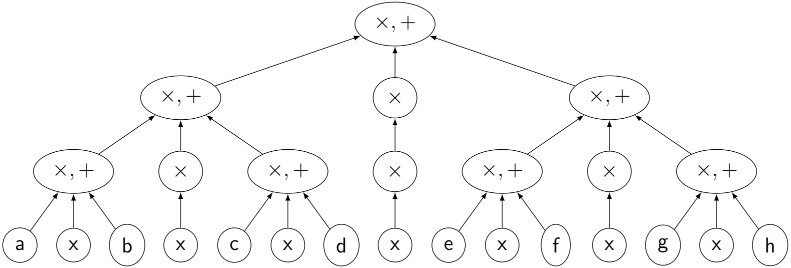
Pipelining: polynomial eval (Estrin's method)

Input:

x, a, b, c, d, e, f, g, h

Compute:

$((ax+b)x^2+(cx+d))x^4+(ex+f)x^2+(gx+h)$



$x^2 = x * x$

$x^4 = x^2 * x^2$

$u = a * x + b$

$v = c * x + d$

$w = e * x + f$

$p = g * x + h$

$q = u * x^2 + v$

$r = w * x^2 + p$

$s = q * x^4 + r \quad \leftarrow$

Pipeline:

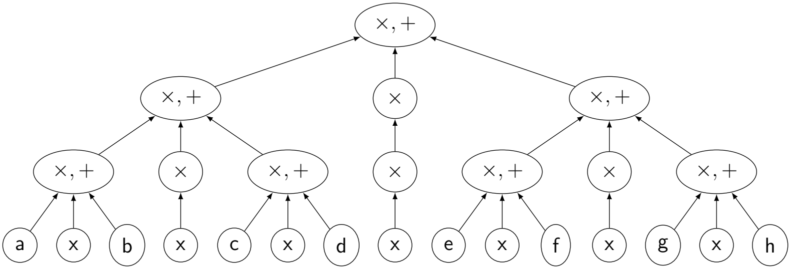
Pipelining: polynomial eval (Estrin's method)

Input:

x, a, b, c, d, e, f, g, h

Compute:

$((ax+b)x^2+(cx+d))x^4+(ex+f)x^2+(gx+h)$



$x^2 = x * x$

$x^4 = x^2 * x^2$

$u = a * x + b$

$v = c * x + d$

$w = e * x + f$

$p = g * x + h$

$q = u * x^2 + v$

$r = w * x^2 + p$

$s = q * x^4 + r \leftarrow$

Pipeline:

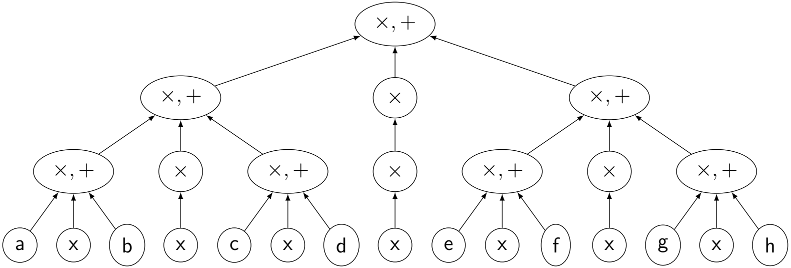
Pipelining: polynomial eval (Estrin's method)

Input:

x, a, b, c, d, e, f, g, h

Compute:

$((ax+b)x^2+(cx+d))x^4+(ex+f)x^2+(gx+h)$



$x^2 = x * x$

$x^4 = x^2 * x^2$

$u = a * x + b$

$v = c * x + d$

$w = e * x + f$

$p = g * x + h$

$q = u * x^2 + v$

$r = w * x^2 + p$

$s = q * x^4 + r$

Pipeline:

14 cycles

$2\times$speedup!

Polynomial evaluation: actual performance

```c
// Horner's rule
for (long i = 0; i < 1000000000L; i++) {
  x = (((((a*x+b)*x+c)*x+d)*x+e)*x+f*x+g)*x+h;
}
```

```
Using Horner's rule:
T = 8.82432
cycles/iter = 29.1203
```

```c
// Estrin's method
for (long i = 0; i < 1000000000L; i++) {
  double x2 = x * x;
  double x4 = x2 * x2;
  x = ((a*x+b)*x2+(c*x+d))*x4+(e*x+f)*x2+(g*x+h);
}
```

```
Using Estrin's method:
T = 5.7813
cycles/iter = 19.0783
```

only $1.5\times$ speedup :(

Polynomial evaluation: actual performance

```
// Horner's rule
for (long i = 0; i < 1000000000L; i++) {
  x = (((((a*x+b)*x+c)*x+d)*x+e)*x+f*x+g)*x+h;
}
```

```
Using Horner's rule:
T = 8.82432
cycles/iter = 29.1203
```

```
// Estrin's method (expanded)
for (long i = 0; i < 1000000000L; i++) {
  double x2 = x * x;
  double x4 = x2 * x2;
  double u = a * x + b;
  double v = c * x + d;
  double w = e * x + f;
  double p = g * x + h;
  double q = u * x2 + v;
  double r = w * x2 + p;
  x = q * x4 + r;
}
```

```
Using Estrin's method:
T = 5.7813
cycles/iter = 19.0783
```

```
Using Estrin's method (expanded):
T = 4.5794
cycles/iter = 15.112
```

$1.9\times$ speedup!

Libraries for special function evaluation

- Baobzi (adaptive fast function interpolator)
  `https://github.com/flatironinstitute/baobzi`
- Agner Fog's Vector Class Library
- SLEEF Vectoried Math Library
- FORTRAN native routines
- C++ Standard Library
- Eigen
- Boost
- AMD Math Library (LibM)
- GNU Scientific Library (GSL)
- Scientific Computing Template Library (SCTL)

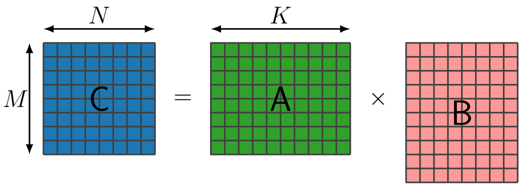
func	name	cycles/eval
bessel_J0	baobzi	20.8
bessel_J0	fort	200.9
bessel_J0	gsl	504.5
bessel_J0	boost	542.9
sin	agnerfog	3.2
sin	sctl	3.6
sin	sleef	4.6
sin	amdlibm	6.9
sin	stl	32.9
sin	eigen	33.1
sin	gsl	149.4

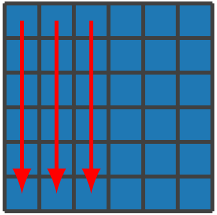
Robert Blackwell - sf_benchmarks :
`https://github.com/flatironinstitute/sf_benchmarks`

FLATIRON INSTITUTE

GEMM micro-kernel

- This is pedagogical – don't write your own GEMM (use BLAS)
- Peak FLOP rate (Skylake core)
  - FMA $(1+1$ per cycle) units $(\times 2)$
  - 512-bit vectors $(\times 8$ for doubles)
  - 3.3GHz clock rate
  - $= 105.6$ GFLOP/s
  - How close can we get to the peak?
- Matrix sizes: M, N, K
- Assume column-major ordering





GEMM micro-kernel

```cpp
template <int M, int N, int K>
void GEMM_ker_naive(double* C, double* A, double* B) {
  for (int k = 0; k < K; k++)
    for (int j = 0; j < N; j++)
      for (int i = 0; i < M; i++)
        C[i+j*M] += A[i+k*M] * B[k+K*j];
}

int main(int argc, char* argv) {
  constexpr int M = 8, N = 8, K = 8;
  double* C = new double[M*N];
  double* A = new double[M*K];
  double* B = new double[K*N];
  // .. init A, B, C

  long L = 1e6;
  double T = -omp_get_wtime();
  for (long i = 0; i < L; i++)
    GEMM_ker_naive<M,N,K>(C, A, B);
  T += omp_get_wtime();
  std::cout<<"FLOP rate = "<<
      2*M*N*K*L/T/1e9 << "GFLOP/s\n";
```

FLATIRON INSTITUTE

GEMM micro-kernel

```
template <int M, int N, int K>
void GEMM_ker_naive(double* C, double* A, double* B) {
  for (int k = 0; k < K; k++)
    for (int j = 0; j < N; j++)
      for (int i = 0; i < M; i++)
        C[i+j*M] += A[i+k*M] * B[k+K*j];
}

int main(int argc, char* argv) {
  constexpr int M = 8, N = 8, K = 8;
  double* C = new double[M*N];
  double* A = new double[M*K];
  double* B = new double[K*N];
  // .. init A, B, C

  long L = 1e6;
  double T = -omp_get_wtime();
  for (long i = 0; i < L; i++)
    GEMM_ker_naive<M,N,K>(C, A, B);
  T += omp_get_wtime();
  std::cout<<"FLOP rate = "<<
      2*M*N*K*L/T/1e9 << "GFLOP/s\n";
```

```
Dimensions: M = N = K = 8

GEMM (naive):
FLOP rate = 5.99578 GFLOP/s
```

FLATIRON INSTITUTE

GEMM micro-kernel

```
template <int M, int N, int K>
void GEMM_ker_vec(double* C, double* A, double* B) {
  using Vec = sctl::Vec<double,M>;

  Vec Cv[N];
  for (int j = 0; j < N; j++)
    Cv[j] = Vec::Load(C+j*M);

  for (int k = 0; k < K; k++) {
    const Vec Av = Vec::Load(A+k*M);
    double* B_ = B + k;
    for (int j = 0; j < N; j++) {
      Cv[j] = Av * B_[K*j] + Cv[j];
    }
  }

  for (int j = 0; j < N; j++)
    Cv[j].Store(C+j*M);
}
```

```
Dimensions: M = N = K = 8

GEMM (naive):
FLOP rate = 5.99578 GFLOP/s
```

FLATIRON INSTITUTE

GEMM micro-kernel

```
template <int M, int N, int K>
void GEMM_ker_vec(double* C, double* A, double* B) {
  using Vec = sctl::Vec<double,M>;

  Vec Cv[N];
  for (int j = 0; j < N; j++)
    Cv[j] = Vec::Load(C+j*M);

  for (int k = 0; k < K; k++) {
    const Vec Av = Vec::Load(A+k*M);
    double* B_ = B + k;
    for (int j = 0; j < N; j++) {
      Cv[j] = Av * B_[K*j] + Cv[j];
    }
  }

  for (int j = 0; j < N; j++)
    Cv[j].Store(C+j*M);
}
```

```
Dimensions: M = N = K = 8

GEMM (naive):
FLOP rate = 5.99578 GFLOP/s

GEMM (vectorized):
FLOP rate = 29.3319 GFLOP/s
```

FLATIRON INSTITUTE

GEMM micro-kernel

```cpp
template <int M, int N, int K>
void GEMM_ker_vec_unrolled(double* C, double* A, double* B) {
  using Vec = sctl::Vec<double,M>;

  Vec Cv[N];
  #pragma GCC unroll (8)
  for (int j = 0; j < N; j++)
    Cv[j] = Vec::Load(C+j*M);

  #pragma GCC unroll (8)
  for (int k = 0; k < K; k++) {
    const Vec Av = Vec::Load(A+k*M);
    double* B_ = B + k;
    #pragma GCC unroll (8)
    for (int j = 0; j < N; j++) {
      Cv[j] = Av * B_[j*K] + Cv[j];
    }
  }

  #pragma GCC unroll (8)
  for (int j = 0; j < N; j++)
    Cv[j].Store(C+j*M);
}
```

```
Dimensions: M = N = K = 8

GEMM (naive):
FLOP rate = 5.99578 GFLOP/s


GEMM (vectorized):
FLOP rate = 29.3319 GFLOP/s
```

FLATIRON INSTITUTE

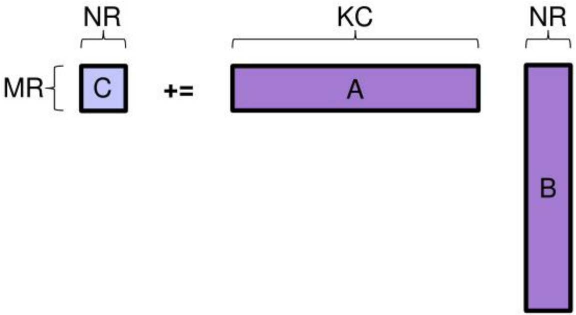
GEMM micro-kernel

```
template <int M, int N, int K>
void GEMM_ker_vec_unrolled(double* C, double* A, double* B) {
  using Vec = sctl::Vec<double,M>;

  Vec Cv[N];
  #pragma GCC unroll (8)
  for (int j = 0; j < N; j++)
    Cv[j] = Vec::Load(C+j*M);

  #pragma GCC unroll (8)
  for (int k = 0; k < K; k++) {
    const Vec Av = Vec::Load(A+k*M);
    double* B_ = B + k;
    #pragma GCC unroll (8)
    for (int j = 0; j < N; j++) {
      Cv[j] = Av * B_[j*K] + Cv[j];
    }
  }

  #pragma GCC unroll (8)
  for (int j = 0; j < N; j++)
    Cv[j].Store(C+j*M);
}
```

```
Dimensions: M = N = K = 8

GEMM (naive):
FLOP rate = 5.99578 GFLOP/s

GEMM (vectorized):
FLOP rate = 29.3319 GFLOP/s

GEMM (vectorized & unrolled):
FLOP rate = 38.5658 GFLOP/s
```

36.5% of peak

GEMM micro-kernel

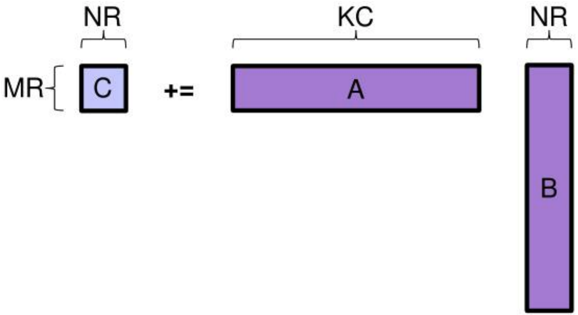


Source: BLIS framework [Van Zee and van de Geijn 2015]

```
Dimensions: M = 8, N = 10, K = 40
```

FLATIRON INSTITUTE

GEMM micro-kernel



Source: BLIS framework [Van Zee and van de Geijn 2015]

```
Dimensions: M = 8, N = 10, K = 40

GEMM (naive):
FLOP rate = 7.9677 GFLOP/s

GEMM (vectorized):
FLOP rate = 65.8419 GFLOP/s

GEMM (vectorized & unrolled):
FLOP rate = 74.9756 GFLOP/s
```

71% of peak!

Instruction-level parallelism – summary

- Modern processors execute a DAG – not a sequence of instructions
  - refactor code to expose instruction parallelism (sometimes extra instructions)
  - loop unrolling, rearranging order of instructions, etc. can help
  - branches can hurt performance – mispredictions have huge penalty
- Primitive data types are vectors – not scalars
  - use SoA data arrangement instead of AoS
  - use vector libraries (VCL, SLEEF, etc) to vectorize code
  - use fast libraries for special functions
- Operations have latency and throughput (pipeline)
  - $+, -, \times$, bitwise operations, etc. are fast
  - other operations are slow
  - aligned memory accesses can be faster
- Resources:
  - Agner Fog: `https://www.agner.org/optimize/`
  - Intel 64 and IA-32 Architectures Optimization Reference Manual

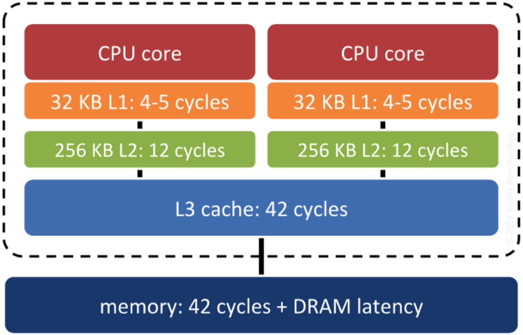
FLATIRON INSTITUTE

Memory

How does computer memory work?

References:

- Ulrich Drepper – What every programmer should know about memory (2007)
- Igor Ostrovsky – Gallery of Processor Cache Effects



Source: Intel Software Developer Manual

Memory benchmarks

```
long N = 1e9; // 8 GB

// Allocate memory
double* X = (double*)malloc(N*sizeof(double));

// Write to array
for (long i = 0; i < N; i++) X[i] = i;

// Update array
for (long i = 0; i < N; i++) X[i] = 2*i;

// Free memory
free(X);
```

FLATIRON INSTITUTE

Memory benchmarks

```
long N = 1e9; // 8 GB

// Allocate memory
double* X = (double*)malloc(N*sizeof(double));

// Write to array
for (long i = 0; i < N; i++) X[i] = i;

// Update array
for (long i = 0; i < N; i++) X[i] = 2*i;

// Free memory
free(X);
```

```
Allocate memory
T = 1.60821e-05

Write to array
T = 1.75352  ---  4.6 GB/s

Update array
T = 0.84467  ---  9.5 GB/s

Free memory
T = 0.0141113
```

Memory benchmarks

```c
long N = 1e9; // 8 GB

// Allocate memory
double* X = (double*)malloc(N*sizeof(double));

// Write to array
for (long i = 0; i < N; i++) X[i] = i;

// Update array
for (long i = 0; i < N; i++) X[i] = 2*i;

// Free memory
free(X);
```

```
Allocate memory
T = 1.60821e-05

Write to array
T = 1.75352   ---   4.6 GB/s

Update array
T = 0.84467   ---   9.5 GB/s

Free memory
T = 0.0141113
```

Memory allocations are not free!

- cost is hidden in initialization (first-touch)

FLATIRON INSTITUTE

Main memory bandwidth

```
long N = 1e9; // 8 GB

// Initialize X, Y
for (long i = 0; i < N; i++) X[i] = Y[i] = i;

// Write to array
#pragma omp parallel for schedule(static)
for (long i = 0; i < N; i++) X[i] = 3.14;

// Read from array
double sum = 0;
#pragma omp parallel for schedule(static) reduction(+:sum)
for (long i = 0; i < N; i++) sum += X[i];

// Adding arrays: 2-reads, 1-write
#pragma omp parallel for schedule(static)
for (long i = 0; i < N; i++) Y[i] += X[i];
```

FLATIRON INSTITUTE

Main memory bandwidth

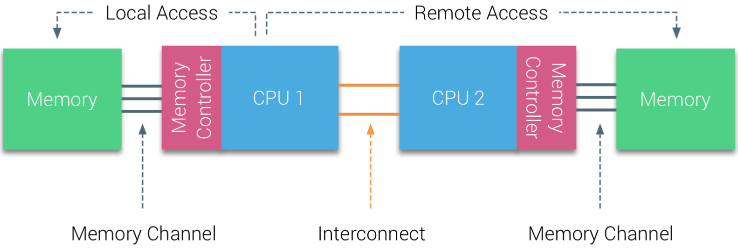
```
long N = 1e9; // 8 GB

// Initialize X, Y
for (long i = 0; i < N; i++) X[i] = Y[i] = i;

// Write to array
#pragma omp parallel for schedule(static)
for (long i = 0; i < N; i++) X[i] = 3.14;

// Read from array
double sum = 0;
#pragma omp parallel for schedule(static) reduction(+:sum)
for (long i = 0; i < N; i++) sum += X[i];

// Adding arrays: 2-reads, 1-write
#pragma omp parallel for schedule(static)
for (long i = 0; i < N; i++) Y[i] += X[i];
```

```
Writing to array
Bandwidth = 35.4136 GB/s

Reading from array
Bandwidth = 69.4623 GB/s

Adding arrays
Bandwidth = 113.637 GB/s
```

FLATIRON INSTITUTE

Non-uniform Memory Access

- Each sockets connected to its own DRAM.
- Sockets interconnected using a network: QPI (Intel), HT (AMD).
- Location of memory pages determined by first-touch policy.



Source: `https://frankdenneman.nl/2016/07/07/numa-deep-dive-part-1-uma-numa`

Main memory bandwidth (NUMA aware)

```
long N = 1e9; // 8 GB

// Initialize X, Y
#pragma omp parallel for schedule(static)
for (long i = 0; i < N; i++) X[i] = Y[i] = i;

// Write to array
#pragma omp parallel for schedule(static)
for (long i = 0; i < N; i++) X[i] = 3.14;

// Read from array
double sum = 0;
#pragma omp parallel for schedule(static) reduction(+:sum)
for (long i = 0; i < N; i++) sum += X[i];

// Adding arrays: 2-reads, 1-write
#pragma omp parallel for schedule(static)
for (long i = 0; i < N; i++) Y[i] += X[i];
```

Set thread affinity:

```
export OMP_PLACES=cores
export OMP_PROC_BIND=spread
```

FLATIRON INSTITUTE

Main memory bandwidth (NUMA aware)

```
long N = 1e9; // 8 GB

// Initialize X, Y
#pragma omp parallel for schedule(static)
for (long i = 0; i < N; i++) X[i] = Y[i] = i;

// Write to array
#pragma omp parallel for schedule(static)
for (long i = 0; i < N; i++) X[i] = 3.14;

// Read from array
double sum = 0;
#pragma omp parallel for schedule(static) reduction(+:sum)
for (long i = 0; i < N; i++) sum += X[i];

// Adding arrays: 2-reads, 1-write
#pragma omp parallel for schedule(static)
for (long i = 0; i < N; i++) Y[i] += X[i];
```

Original:

```
Writing to array
Bandwidth = 35.4136 GB/s

Reading from array
Bandwidth = 69.4623 GB/s

Adding arrays
Bandwidth = 113.637 GB/s
```

NUMA aware:

```
Writing to array
Bandwidth = 87.1515 GB/s

Reading from array
Bandwidth = 160.663 GB/s

Adding arrays
Bandwidth = 180.069 GB/s
```

FLATIRON INSTITUTE

Main memory bandwidth (NUMA aware)

Many shared-memory codes scale poorly because they don't account for NUMA!

Original:

```
Writing to array
Bandwidth = 35.4136 GB/s

Reading from array
Bandwidth = 69.4623 GB/s

Adding arrays
Bandwidth = 113.637 GB/s
```

NUMA aware:

```
Writing to array
Bandwidth = 87.1515 GB/s

Reading from array
Bandwidth = 160.663 GB/s

Adding arrays
Bandwidth = 180.069 GB/s
```

FLATIRON INSTITUTE

L1-cache bandwidth

```
long N = 2048; // 16KB
double* X = (double*)malloc(N*sizeof(double));
double* Y = (double*)malloc(N*sizeof(double));
// Initialize X, Y

// Write to array
for (long i = 0; i < N; i++) X[i] = 3.14;

// Read from array
double sum = 0;
for (long i = 0; i < N; i++) sum += X[i];

// Adding arrays: 2-reads, 1-write
for (long i = 0; i < N; i++) Y[i] += X[i];
```

FLATIRON INSTITUTE

L1-cache bandwidth

```c
long N = 2048; // 16KB
double* X = (double*)malloc(N*sizeof(double));
double* Y = (double*)malloc(N*sizeof(double));
// Initialize X, Y

// Write to array
for (long i = 0; i < N; i++) X[i] = 3.14;

// Read from array
double sum = 0;
for (long i = 0; i < N; i++) sum += X[i];

// Adding arrays: 2-reads, 1-write
for (long i = 0; i < N; i++) Y[i] += X[i];
```

```
Writing to array
Bandwidth = 26.2744 GB/s

Reading from array
Bandwidth = 6.57305 GB/s

Adding arrays
Bandwidth = 131.203 GB/s
```

FLATIRON INSTITUTE

L1-cache bandwidth (vectorized)

```
using Vec = sctl::Vec<double,8>;

long N = 2048; // 16KB
double* X = (double*)malloc(N*sizeof(double));
double* Y = (double*)malloc(N*sizeof(double));
// Initialize X, Y

// Write to array
Vec v = 3.14;
#pragma GCC unroll (4)
for (long i = 0; i < N; i+=8) v.Store(X+i);

```

FLATIRON INSTITUTE

L1-cache bandwidth (vectorized)

```
using Vec = sctl::Vec<double,8>;

long N = 2048; // 16KB
double* X = (double*)malloc(N*sizeof(double));
double* Y = (double*)malloc(N*sizeof(double));
// Initialize X, Y

// Write to array
Vec v = 3.14;
#pragma GCC unroll (4)
for (long i = 0; i < N; i+=8) v.Store(X+i);

```

```
Writing to array
Bandwidth = 89.5993 GB/s
cycles/iter = 2.35716
```

FLATIRON INSTITUTE

L1-cache bandwidth (vectorized)

```
// Read from array
Vec sum[8] = {0.,0.,0.,0.,0.,0.,0.,0.};
for (long i = 0; i < N; i+=8*8) {
  sum[0] = sum[0] + Vec::Load(X   +i);
  sum[1] = sum[1] + Vec::Load(X+8 +i);
  sum[2] = sum[2] + Vec::Load(X+16+i);
  sum[3] = sum[3] + Vec::Load(X+24+i);
  sum[4] = sum[4] + Vec::Load(X+32+i);
  sum[5] = sum[5] + Vec::Load(X+40+i);
  sum[6] = sum[6] + Vec::Load(X+48+i);
  sum[7] = sum[7] + Vec::Load(X+56+i);
}
```

```
Writing to array
Bandwidth = 89.5993 GB/s
cycles/iter = 2.35716
```

FLATIRON INSTITUTE

L1-cache bandwidth (vectorized)

```
// Read from array
Vec sum[8] = {0.,0.,0.,0.,0.,0.,0.,0.};
for (long i = 0; i < N; i+=8*8) {
  sum[0] = sum[0] + Vec::Load(X   +i);
  sum[1] = sum[1] + Vec::Load(X+8 +i);
  sum[2] = sum[2] + Vec::Load(X+16+i);
  sum[3] = sum[3] + Vec::Load(X+24+i);
  sum[4] = sum[4] + Vec::Load(X+32+i);
  sum[5] = sum[5] + Vec::Load(X+40+i);
  sum[6] = sum[6] + Vec::Load(X+48+i);
  sum[7] = sum[7] + Vec::Load(X+56+i);
}
```

```
Writing to array
Bandwidth = 89.5993 GB/s
cycles/iter = 2.35716

Reading from array
Bandwidth = 210.375 GB/s
cycles/iter = 1.00392
```

L1-cache bandwidth (vectorized)

```

// Adding arrays: 2-reads, 1-write
for (long i = 0; i < N; i+=8*2) {
  Vec X0 = Vec::Load(X+0+i);
  Vec X1 = Vec::Load(X+8+i);
  Vec Y0 = Vec::Load(Y+0+i);
  Vec Y1 = Vec::Load(Y+8+i);
  (X0+Y0).Store(Y+VecLen*0+i);
  (X1+Y1).Store(Y+VecLen*1+i);
}
```

```
Writing to array
Bandwidth = 89.5993 GB/s
cycles/iter = 2.35716

Reading from array
Bandwidth = 210.375 GB/s
cycles/iter = 1.00392
```

FLATIRON INSTITUTE

L1-cache bandwidth (vectorized)

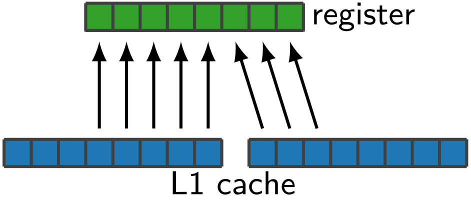
```
// Adding arrays: 2-reads, 1-write
for (long i = 0; i < N; i+=8*2) {
  Vec X0 = Vec::Load(X+0+i);
  Vec X1 = Vec::Load(X+8+i);
  Vec Y0 = Vec::Load(Y+0+i);
  Vec Y1 = Vec::Load(Y+8+i);
  (X0+Y0).Store(Y+VecLen*0+i);
  (X1+Y1).Store(Y+VecLen*1+i);
}
```

```
Writing to array
Bandwidth = 89.5993 GB/s
cycles/iter = 2.35716

Reading from array
Bandwidth = 210.375 GB/s
cycles/iter = 1.00392

Adding arrays
Bandwidth = 148.29 GB/s
cycles/iter = 4.27271
```

FLATIRON INSTITUTE

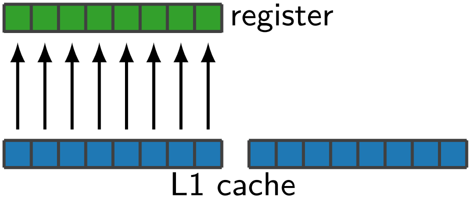
L1-cache bandwidth (vectorized & aligned)

Unaligned read:



L1-cache bandwidth (vectorized & aligned)

Aligned read:

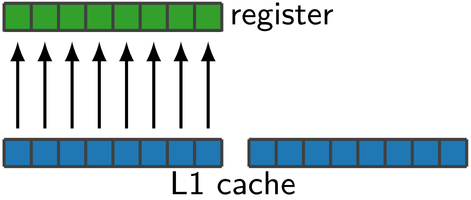


Replace:

- malloc → sctl::aligned_new
- Vec::Load → Vec::AlignedLoad
- Vec::Store → Vec::AlignedStore

L1-cache bandwidth (vectorized & aligned)

Aligned read:




```
Writing to array
Bandwidth = 210.273 GB/s
cycles/iter = 1.00441

Reading from array
Bandwidth = 380.953 GB/s
cycles/iter = 0.554399

Adding arrays
Bandwidth = 325.592 GB/s
cycles/iter = 1.94599
```

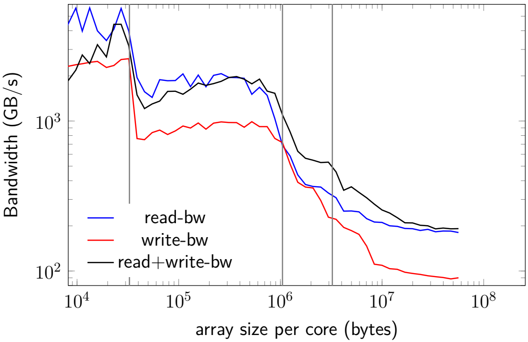
Replace:

- malloc → sctl::aligned_new
- Vec::Load → Vec::AlignedLoad
- Vec::Store → Vec::AlignedStore

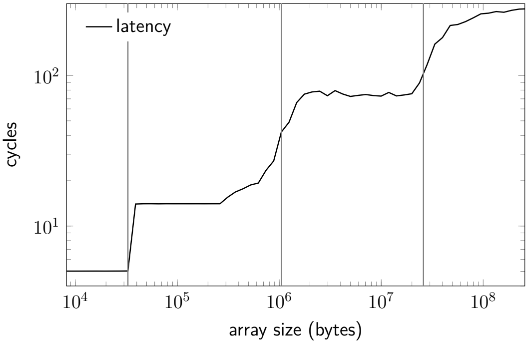
Aligned memory acceses to L1 can be $2\times$ faster!

Memory bandwidth and latency

$32\times$ difference between
L1 and main memory bandwidth!

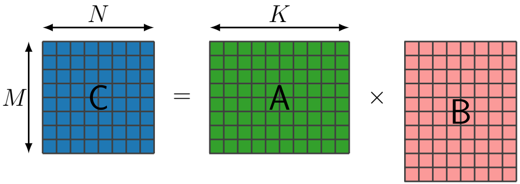


$56\times$ difference between
L1 and main memory latency!



FLATIRON INSTITUTE

Optimizing GEMM for memory access



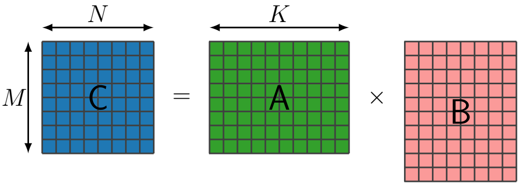

```
void GEMM(int M, int N, int K, double* A, int LDA,
          double* B, int LDB, double* C, int LDC) {
  for (int j = 0; j < N; j++)
    for (int k = 0; k < K; k++)
      for (int i = 0; i < M; i++)
        C[i+j*LDC] += A[i+k*LDA] * B[k+j*LDB];
}
```

Dimensions: M = N = K = 2000

perf: performance monitoring tool which samples hardware counters

FLATIRON INSTITUTE

Optimizing GEMM for memory access




```
void GEMM(int M, int N, int K, double* A, int LDA,
          double* B, int LDB, double* C, int LDC) {
  for (int j = 0; j < N; j++)
    for (int k = 0; k < K; k++)
      for (int i = 0; i < M; i++)
        C[i+j*LDC] += A[i+k*LDA] * B[k+j*LDB];
}
```

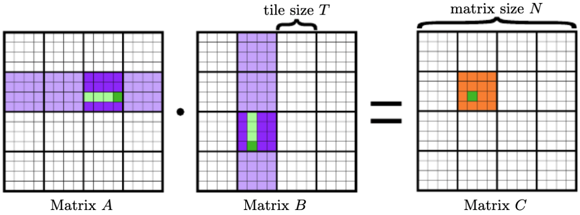
Dimensions: $M = N = K = 2000$

perf: performance monitoring tool which samples hardware counters

```
~> g++ -O3 -march=native gemm.cpp
~> perf stat -e L1-dcache-load-misses \
   -e L1-dcache-loads -e l2_rqsts.miss \
   -e l2_rqsts.references -e LLC-load-misses \
   -e LLC-loads ./a.out

FLOP rate = 4.87547 GFLOP/s

30,311,624,911  L1-dcache-loads
14,900,283,807  L1-dcache-load-misses   49.16% of all L1-
24,387,281,512  l2_rqsts.references
10,034,752,513  l2_rqsts.miss
 2,260,778,457  LLC-loads
 1,310,606,484  LLC-load-misses         57.97% of all LL-
```

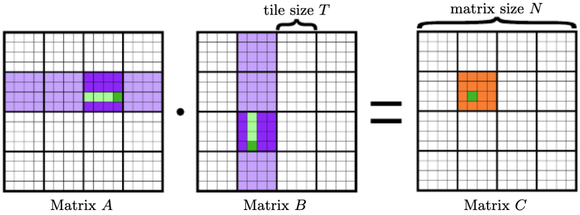
GEMM blocking

```
template <int M, int N, int K>
void GEMM_blocked(double* A, int LDA,
        double* B, int LDB, double* C, int LDC) {
  for (int j = 0; j < N; j++)
    for (int k = 0; k < K; k++)
      for (int i = 0; i < M; i++)
        C[i+j*LDC] += A[i+k*LDA] * B[k+j*LDB];
}

template <int M, int N, int K,
          int Mb, int Nb, int Kb, int... NN>
void GEMM_blocked(double* A, int LDA,
        double* B, int LDB, double* C, int LDC) {
  for (int j = 0; j < N; j+=Nb)
    for (int i = 0; i < M; i+=Mb)
      for (int k = 0; k < K; k+=Kb)
        GEMM_blocked<Mb,Nb,Kb, NN...>(A+i+k*LDA,LDA,
                              B+k+j*LDB,LDB, C+i+j*LDC,LDC);
}
```



GEMM blocking

```cpp
template <int M, int N, int K>
void GEMM_blocked(double* A, int LDA,
        double* B, int LDB, double* C, int LDC) {
  GEMM_ker_vec_unrolled<M,N,K>(A,LDA, B,LDB, C,LDC);
}




template <int M, int N, int K,
          int Mb, int Nb, int Kb, int... NN>
void GEMM_blocked(double* A, int LDA,
        double* B, int LDB, double* C, int LDC) {
  for (int j = 0; j < N; j+=Nb)
    for (int i = 0; i < M; i+=Mb)
      for (int k = 0; k < K; k+=Kb)
        GEMM_blocked<Mb,Nb,Kb, NN...>(A+i+k*LDA,LDA,
                               B+k+j*LDB,LDB, C+i+j*LDC,LDC);
}
```



GEMM blocking

```cpp
template <int M, int N, int K>
void GEMM_blocked(double* A, int LDA,
        double* B, int LDB, double* C, int LDC) {
  GEMM_ker_vec_unrolled<M,N,K>(A,LDA, B,LDB, C,LDC);
}




template <int M, int N, int K,
          int Mb, int Nb, int Kb, int... NN>
void GEMM_blocked(double* A, int LDA,
        double* B, int LDB, double* C, int LDC) {
  for (int j = 0; j < N; j+=Nb)
    for (int i = 0; i < M; i+=Mb)
      for (int k = 0; k < K; k+=Kb)
        GEMM_blocked<Mb,Nb,Kb, NN...>(A+i+k*LDA,LDA,
                              B+k+j*LDB,LDB, C+i+j*LDC,LDC);
}
```

GEMM_blocked<M,N,K, 8,10,40>(...)

```
FLOP rate = 11.803 GFLOP/s
11,514,598,988  L1-dcache-loads
 3,274,256,252  L1-dcache-load-misses   28.44% of all L1-
 3,283,717,404  l2_rqsts.references
 1,047,408,896  l2_rqsts.miss
 1,032,604,200  LLC-loads
   293,256,535  LLC-load-misses         28.40% of all LL-
```

GEMM blocking

```
template <int M, int N, int K>
void GEMM_blocked(double* A, int LDA,
        double* B, int LDB, double* C, int LDC) {
  GEMM_ker_vec_unrolled<M,N,K>(A,LDA, B,LDB, C,LDC);
}




template <int M, int N, int K,
          int Mb, int Nb, int Kb, int... NN>
void GEMM_blocked(double* A, int LDA,
        double* B, int LDB, double* C, int LDC) {
  for (int j = 0; j < N; j+=Nb)
    for (int i = 0; i < M; i+=Mb)
      for (int k = 0; k < K; k+=Kb)
        GEMM_blocked<Mb,Nb,Kb, NN...>(A+i+k*LDA,LDA,
                      B+k+j*LDB,LDB, C+i+j*LDC,LDC);
}
```

GEMM_blocked<M,N,K, 8,10,40>(...)

```
FLOP rate = 11.803 GFLOP/s
11,514,598,988   L1-dcache-loads
 3,274,256,252   L1-dcache-load-misses   28.44% of all L1-
 3,283,717,404   l2_rqsts.references
 1,047,408,896   l2_rqsts.miss
 1,032,604,200   LLC-loads
   293,256,535   LLC-load-misses         28.40% of all LL-
```

GEMM_blocked<M,N,K, 40,40,40, 8,10,40>(...)

```
FLOP rate = 26.5831 GFLOP/s
11,533,695,903   L1-dcache-loads
 1,084,624,171   L1-dcache-load-misses    9.40% of all L1-
 1,091,155,596   l2_rqsts.references
   538,256,077   l2_rqsts.miss
   470,615,736   LLC-loads
   112,816,293   LLC-load-misses         23.97% of all LL-
```

FLATIRON INSTITUTE

GEMM blocking

```cpp
template <int M, int N, int K>
void GEMM_blocked(double* A, int LDA,
        double* B, int LDB, double* C, int LDC) {
  GEMM_ker_vec_unrolled<M,N,K>(A,LDA, B,LDB, C,LDC);
}




template <int M, int N, int K,
          int Mb, int Nb, int Kb, int... NN>
void GEMM_blocked(double* A, int LDA,
        double* B, int LDB, double* C, int LDC) {
  for (int j = 0; j < N; j+=Nb)
    for (int i = 0; i < M; i+=Mb)
      for (int k = 0; k < K; k+=Kb)
        GEMM_blocked<Mb,Nb,Kb, NN...>(A+i+k*LDA,LDA,
                              B+k+j*LDB,LDB, C+i+j*LDC,LDC);
}
```

GEMM_blocked<M,N,K, 40,40,40, 8,10,40>(...)

```
FLOP rate = 26.5831 GFLOP/s
11,533,695,903  L1-dcache-loads
 1,084,624,171  L1-dcache-load-misses     9.40% of all L1-
 1,091,155,596  l2_rqsts.references
   538,256,077  l2_rqsts.miss
   470,615,736  LLC-loads
   112,816,293  LLC-load-misses          23.97% of all LL-
```

FLATIRON INSTITUTE

GEMM blocking

```
template <int M, int N, int K>
void GEMM_blocked(double* A, int LDA,
        double* B, int LDB, double* C, int LDC) {
  GEMM_ker_vec_unrolled<M,N,K>(A,LDA, B,LDB, C,LDC);
}




template <int M, int N, int K,
          int Mb, int Nb, int Kb, int... NN>
void GEMM_blocked(double* A, int LDA,
        double* B, int LDB, double* C, int LDC) {
  for (int j = 0; j < N; j+=Nb)
    for (int i = 0; i < M; i+=Mb)
      for (int k = 0; k < K; k+=Kb)
        GEMM_blocked<Mb,Nb,Kb, NN...>(A+i+k*LDA,LDA,
                              B+k+j*LDB,LDB, C+i+j*LDC,LDC);
}
```

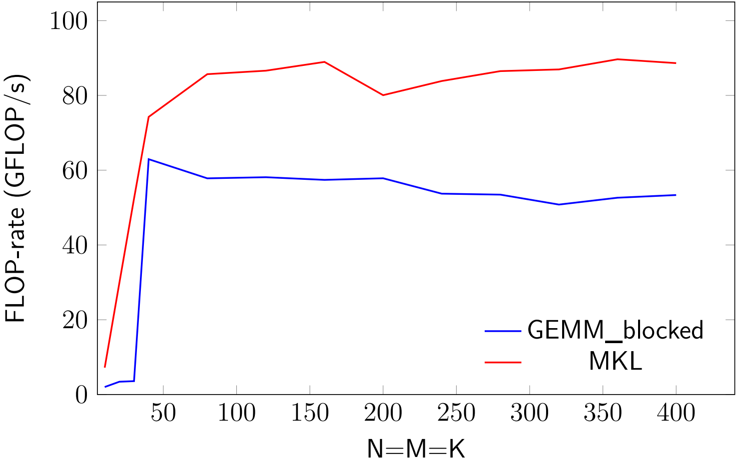
GEMM_blocked<M,N,K, 40,40,40, 8,10,40>(...)

```
FLOP rate = 26.5831 GFLOP/s
11,533,695,903  L1-dcache-loads
 1,084,624,171  L1-dcache-load-misses    9.40% of all L1-
 1,091,155,596  l2_rqsts.references
   538,256,077  l2_rqsts.miss
   470,615,736  LLC-loads
   112,816,293  LLC-load-misses         23.97% of all LL-
```

GEMM_blocked<M,N,K, 200,200,200, 40,40,40, 8,10,40>(...)

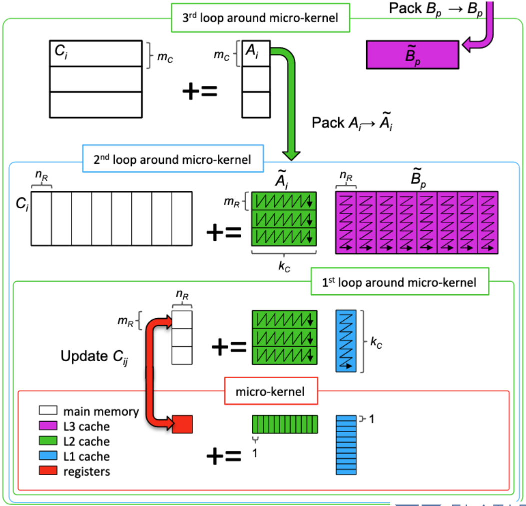
```
FLOP rate = 43.1604 GFLOP/s
11,531,903,350  L1-dcache-loads
 1,094,841,388  L1-dcache-load-misses    9.49% of all L1-
 1,194,502,755  l2_rqsts.references
   201,888,454  l2_rqsts.miss
   116,940,584  LLC-loads
    44,894,302  LLC-load-misses         38.39% of all LL-
```

GEMM benchmarks



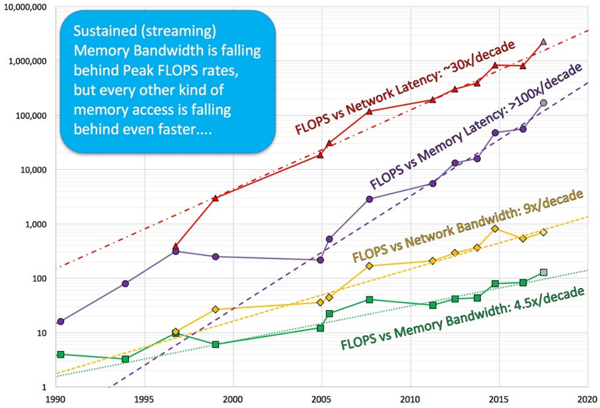
Optimizing GEMM – references

BLIS framework:
Van Zee and van de Geijn 2015



Memory and caches – summary

- Memory bandwidth and latency are lagging behind FLOP rates
- Latency is a bigger issue: avoid linked lists, pointer chasing, etc. — use arrays, regular memory accesses instead
- Caches are fast - use them optimally
- Account for NUMA
- New technologies (HBM) are probably on the way



Source: John McCalpin - Memory bandwidth and system balance in HPC systems, 2016